Goal-oriented requirement analysis for process control system design

El-Maddah, Islam Ahmed Mahmoud

Goal-Oriented Requirements Analysis for Process Control Systems

Islam Ahmed Mahmoud El-Maddah

A thesis submitted to the University of London
in partial fulfilment of the Requirements for the degree of
Doctor of Philosophy in Computer Science

King's College London
Department of Computer Science

2004

Abstract

The research addresses the field of software development of process control applications. The B formal method has been demonstrated to provide effective support at the early design stages. It has been constructively used in building specifications of process control systems.

However, the formal method assumes correct, complete, consistent and user-agreed formal requirements; this is difficult to achieve without a potential interaction between the systems engineer, as the client, and the software engineer, as the contractor, which entails concern-interference between the two and a detailed awareness of formal methods from the systems engineer. This indicates an existing gap between the systems engineer's perspective and the formal specification level.

The research attempts to fill this gap and focuses on the early requirements stages, which have a direct impact on the success of implementing the executable programs; the research focuses on automating the requirements analysis stage. This automation is achieved by implementing an interactive software tool, GOPCSD.

We adopted the KAOS method to structure and analyse the requirements; this provides the GOPCSD tool with its goal-oriented nature that enables refining the high-level user needs into low-level operational goals.

However, unlike the general KAOS method, the GOPCSD method was implemented to support process control systems. This motivated us to specialise and extend the KAOS method and to construct a library of process control requirements where the details of the frequently-used components and the abstract high-level functions of the process control applications are stored to be reused in similar applications.

In particular, we identified new refinement patterns, which were shown to be helpful in constructing process control applications. Furthermore, we extended the checks the user can apply to the goal-models by adding completeness, animation, and reachability tests.

After modifying the requirements, the GOPCSD tool automatically generates a B specification, which can be further processed by a software engineer within the B toolkit environment.

Two case studies of a gas burner and production cell are presented to examine the GOPCSD method and assess its supporting tool. Finally, we compare the specifications of the GOPCSD tool with other related methods.

I dedicate my thesis to the spirits of my parents:
Fawkia and Ahmed
I believe that it is your honour before me to achieve this.
Thanks for teaching me
"I can accomplish my goals by my faith in GOD"
I believe you are sharing these moments with me.
I will always keep my promise to both of you.

Always in my heart

Acknowledgement

"Only those who are thankful will receive more"

To begin with, I thank GOD for his generosity and aid; I ask HIM to keep guiding me to utilise what HE has blessed me with and giving me the patience to pass through difficult times, the strength and solidarity to accomplish my goals, and the wisdom to decide on my directions.
"My Lord, we do not have any knowledge but that you gave us"

I would like to thank Prof. Maibaum for his support and his faith in my abilities. Thanks Tom for being such a nice friend and a very helpful and guiding supervisor. I think it was my good luck to have you as my supervisor. Thanks again for giving me all these research values. I will try my best to keep them.

I gratefully acknowledge financial support by the Egyptian government, which enabled me to complete my postgraduate studies at King's.

Many thanks to my department, Computer and Systems Engineering at Ain Shams University, Cairo, Egypt, where I received my Bachelors and Masters degrees of Science and learnt about Engineering and Computing.

I would like to thank Dr. Lano for his guidance and valuable comments and allowing me opportunities for fruitful discussions.

I would like to gratefully thank all the researchers whom I have referenced in my thesis. Thanks for all this work that guided and helped me and other people. I hope I could help others as well.

Great thanks for all the members of my research group, the "Software Engineering Group", for all these beneficial seminars and discussions we have had together.

Special "Thank You" to my Brother Ehab and my Sisters Enas and Eman for their encouragement and support, even remotely.

I would like to thank my dear colleague Marhung for her encouragement.

Last but not least, I would like to thank my colleagues in the Computer Science Department at King's for the nice time and the good memories I will always keep in my heart.

Table of Contents

List of Figures

List of Tables

Introduction 1

This chapter is an introduction to the thesis; we highlight the related research areas; then, we introduce the research objectives. Afterwards, we briefly outline the thesis structure. At the end of the chapter, the research hypothesis and the main contributions are stated in the last section.

1.1 Introduction

Within the last two decades, much effort has been focused on automating the generation process of software applications. Automatically generating programs involves the use of software tools, which are capable of interacting with the user at different levels to translate the user inputs into code. Automation may be introduced at various stages of the development lifecycle, ranging from support for requirements engineering to code generation. Such tools should possess the capability of analysing the user inputs to enhance them, as well as generating executable programs, formal specifications, or requirements. In addition to time and effort reduction, the chance of deviating the application specifications from the initial requirements will decrease, as will the potential hazards during the run-time of the software programs.

In order to obtain maximum gain from automating a software application development method, the method should have a narrow scope, reducing the effort expended by the user in modelling the particular family of applications [Maibaum Y2K]. Having a specific domain can encourage a deeper level of reusability and the building of a requirements or specification library [Robertson and Robertson 99]. Hence, the development method can be tailored to fit an application family, as well as communicating with the user in a language and terminologies closer to his/her perspective. The tool itself can translate the user input, after some processing, to a different form. In particular, the (process) systems engineer can be guided to develop requirements using the terminology and components of the

domain and the tool can then translate these requirements to formal specifications, ready for processing by a software engineer. Moreover, focusing on a narrow family of applications increases both the hardware and software reusability in building similar applications within the same family. The research focuses on Process Control Systems such as burner systems, hydraulic systems, production cells, and lift systems. These process control systems are regarded as Reactive Systems [Mellor 85, Wieringa 01] that communicate with their local environment through sensors and actuators. Such communication is continuous: the system senses a number of variables and then reacts to changes in them, producing one or more actions that affect the local environment. And hence, the environment affects the system by changing the input variables.

As the requirements gathering and checking phase is very important to eliminate bugs that are often detected during the design or implementation phases [Davis 93], resulting in costly repair work, the research starts from this requirements phase and provides considerable checking and validation analysis.

Creating a software controller for process control systems generally involves two preliminary stages: eliciting system requirements and formally specifying the software programs to be constructed. This entails co-operation and knowledge exchange between a systems engineer(s) and a software engineer(s); these two engineers usually play the role of the client and the contractor, as shown in figure 1.1, part (a). The systems engineer is supposed to have complete information about the operations of the process control system that makes him/her capable of providing the requirements to the Software Engineer. These requirements are then processed by the Software Engineer, and finally translated to formal specifications or design documents: in case of formal specifications, the Software Engineer should use a Formal Method [Hinchey 95], like B [Abrial 95, Wordsworth 96] or Z [Wordsworth 92], which has the capability of translating the specifications into an executable version that can control the process control system ensuring that the implementation version will exhibit the same properties as stated at the specification level. Or, as another alternative, the software engineer codes executable programs that satisfy the generated design.

However, as in [Cholvy at al. 02], formal methods, such as those referred to above, are not suitable to serve as the initial stage of creating the software applications. They usually presume that user needs are understood satisfactorily and proceed to formalise and refine these user needs. Thus, there is an implicit assumption that the user agrees with such refinements even before testing or validating such refinements. The interaction between the systems engineer, who provides abstract and informal requirements, and the software engineer, who attempts to formalise and refine these requirements engenders interference of concerns.

For example, the available requirements may still exhibit incompleteness, inconsistency or unreachable states in behaviour as follows:

- The systems engineer may provide different levels of operational requirements that can override each other, and as a result producing inconsistency and/or ambiguity that have to be resolved by the Software Engineer.
- It is difficult for the systems engineer to correctly and precisely express the details of the intended operations in terms of the intended sequences of events, which are required by the software

engineer. Some parts of the requirements may be missed out, for example, what happens under some specific combinations of the inputs.

- Sometimes, the systems engineer may locally specify the operations disregarding the special cases or other interfering requirements.

Such reasons can lead to inconsistency and incompleteness in the requirements, which in turn results in hazards during the operation of the system. Moreover, the software engineer will usually be in charge of taking modification decisions when checking the specifications (for example, when animating or verifying the specifications). He/she may not be the right person to be making the decisions about the appropriate fixes to be made.

Similar to the concept of "*divide and conquer*", the hierarchical nature of the KAOS [VanLamsweerde 91] goal-model starts with abstract overall goals and then focuses on decomposed, less abstract parts of the requirements, proceeding to further parts until completing the whole goal-model. In addition to dividing the effort, the hierarchical nature of the goal-model addresses another important issue, which is that the formality gradually increases downwards (from the top-level goal to the terminal goals). Thus, as shown in figure 1.1, narrowing the gap between the systems engineer's perspective and the formal specification level used by the software engineer can be accomplished through the goal-driven requirements analysis of KAOS, after some adaptations to suit the nature of process control systems.

Figure 1.1, part (b), shows how the insertion of the GOPCSD (**G**oal **O**riented **P**rocess **C**ontrol **S**ystems **D**esign) tool will fill the large gap between the systems engineer and the formal specification; instead of the normal, unstructured co-operation between the software and systems engineers, which leads to concern over interference. It is important to mention that the GOPCSD tool will not eliminate the role of the software engineer in the development lifecycle, but delay it to processing the generated formal specifications. The software engineer will be able to use a formal method's development environment, like the B toolkit [Lano and Haughton 96, Wordworth, 96], to refine/animate/prove the specifications and finally, produce implementation programs.

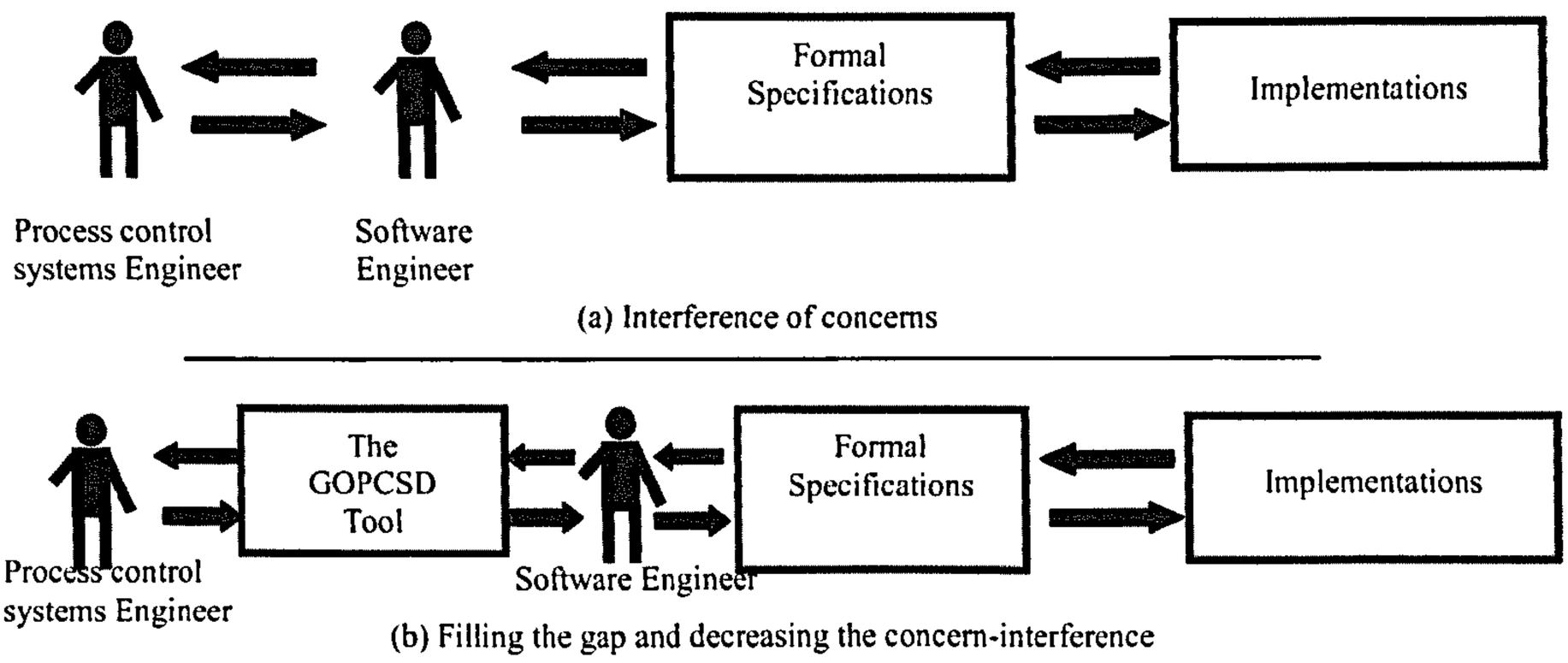


Figure 1.1, the system and software engineers' interaction within the development lifecycle

In order to evaluate the developed method and tool and to judge how suitable it is from the systems engineer's perspective, we will proceed as follows. We will develop two general process

control case studies, often used in the software engineering and formal methods literature. Then, we will rely on our experience in the field of control engineering to enable us to qualitatively assess the effectiveness of the method and tool. This experience in the field of control engineering will provide a background for analysing the requirements not only from the perspective of the software engineering expert, but also from that of the experienced systems engineer.

1.2 Research Objectives

This Research combines the B formal method and the goal-oriented requirements analysis method of KAOS to serve software development by specifying controllers for process control systems. The objective of this research is to separate the concerns of the software and systems engineers and to guide the systems engineer in designing process control applications. This is aided by hiding from the systems engineer the high-level of logical or mathematical knowledge required by the software engineer in order to do detailed software design. From an alternative viewpoint, the GOPCSD method should aid the systems engineer to specify, reason about, check, validate and modify the requirements without specifying how to achieve them (by means of detailed design of software).

The research will reduce this interference of concerns between the systems and the software engineers through delaying the activities related to software until having generated the preliminary B specifications. To be able to judge the suitability and the success of the method, we will develop an interactive tool, which supports an extension of the KAOS method.

The GOPCSD tool supposes that the user will be a systems engineer with general expertise in process-control applications and strong awareness of the process-control applications to be developed, but with little knowledge about formal methods. The Systems Engineer should be capable of specifying the application requirements through the GOPCSD tool in order for the latter to translate the requirements into B specification machines, which can be processed by a software engineer within the B formal method environment (B toolkit). Therefore, we should expect the following features from the GOPCSD tool:

- Helping the systems engineer to develop a better understanding of the requirements.

- Generating well-documented specifications, tracing the corrected requirements to the specification level for further processing by the software engineer; and hence, using this requirements model to easily change/evolve the applications when required.

- Decreasing the interference of concerns between the systems and software engineers and providing a library of requirements reuse to shorten the development time.

- Assisting the systems engineer in predicting obstacles, detecting goal-conflicts, and resolving them by enabling the user to modify the requirements.

- Providing a chance for the systems engineer to validate the requirements, especially after modifying parts of them as a response to tool generated suggestions.

1.3 Thesis Outline

This thesis consists of nine chapters and two appendices. We establish the research areas in this chapter. In chapter two, we introduce the closed-loop and discrete control concepts, since the research problem is mainly that of building a discrete-time controller. The research problem is formally defined in addition to the basic assumptions about the related software and process systems engineers. Following that, we fix the relevant research boundaries. Chapter two ends by surveying the related research for building specifications for process control systems.

In Chapter three, we highlight the Requirements Engineering aspects. And hence, we present the KAOS method as a goal-oriented requirement analysis method. Then, we end chapter three by addressing the other goal-oriented requirements analysis research.

Although the GOPCSD hides the details of the B formal method, we illustrate the role of formal methods in chapter four with more stress on the B method; furthermore, we provide a study of a gas burner system to exemplify how to specify process-control Systems in the B AMN language. We will use this example later in chapter seven as the first case study, in order to extract the similarities and differences between the different design approaches.

The GOPCSD method is described in detail in chapter five, where we introduce the KAOS method adaptations required to fit process control systems features. The chapter points to the motivations of building a library with the frequently used process control applications' requirements. Finally, we provide the semantics required to build the GOPCSD tool algorithms.

In chapter six, we illustrate the detailed functions of the GOPCSD tool. The main constructs of the tool like goal, agent and variable, are described as well as the ideas about importing, refining, combining, and checking the requirements' parts.

In chapters seven and eight, we examine two case studies: a gas burner system and a production cell, respectively, to evaluate the tool and illustrate the method and the use of the GOPCSD tool. We portray the construction of goal-oriented requirements, as well as the possible modifications after checking and animating the requirements. Finally, we provide the generated B specifications for the two applications.

In chapter nine, a discussion of the work and conclusions are presented. Further, we provide suggestions for possible future work and some possible extensions of the tool.

Appendix A is the documentation of the tool; it contains a brief analysis and design report for the GOPCSD tool. In addition, we provide a user guide with screenshots from the GOPCSD tool to illustrate how the tool works. Appendix B contains the details of the requirements' goal-models, their checks' results and the listing of the B machines of the gas burner and production cell applications presented in chapters seven and eight.

1.4 Main Contributions

In this research, we designed and developed an automated early requirements analysis tool, which adapts the KAOS method to fit process control systems. Our main contributions and innovations can be summarised as follows:

- **Reducing the interference of concerns between the systems and software engineers. Automating the early requirements stages of the development of process control applications.** Guiding the systems engineer to take the requirements decision before translating the requirements into formal specification, and hence preparing the stage for the software engineer to be in charge for developing the application.
- **Validating the requirements by providing animation utilities to portray the application performance at run-time.** This animation utility enables the user to make an early judgement about the refined requirements and, hence, capture most of the requirements bugs and enhance the application's likely success at this early stage.
- **Migrating reusability of the process control applications to the requirements level.** A library of the most common components and the high-level abstract functions of the various process control application was implemented; the systems engineer can add new elements to it.
- **Identifying refinement patterns for reactive systems in general and particularly for process control systems.** These patterns extracted from process control applications provide better understanding of the requirements model and guide the construction of the goal-model, the formalisation and translation to B machines.
- **Enabling the development of more than one version of the application.** This is achieved by allowing the goal-model to contain goal-refinement sites of alternative type in a compact goal-model to represent multiple solutions. Afterwards, the different simple goal-model versions can be generated by splitting the compact one.

The Research Problem 2

Since the research problem involves designing a software controller, we proceed in this chapter with a brief discussion of the basic concepts of control engineering. Then, we introduce the research problem, as well as defining the limitations and boundaries of the research. Finally, we describe the existing related approaches for specifying process control systems or reactive systems in general.

2.1 Control Systems

Controlling parts of the environment, as an activity, dates back to the earliest times since humans have been able to devise and utilise tools. The main objective of control activities is to induce, by directing some activity, the output(s) corresponding to a specific value or range of values of the input applied to the activity. Before controlling a process, a system, or local environment, a good understanding of the controlled process itself should be available, as well identifying the inputs and the outputs of the process. The controlling activity can be regarded as introducing a sub-system that controls the existing system in order to adapt the overall performance and/or input/output characteristics. A mathematical model or formal specification should be developed for the process and for the overall system behaviour, especially as cause-effect (possible set of *"IF THEN"*) rules between the input(s) and the output(s).

The controller sub-system may be composed of hardware (mechanical, electrical or pneumatic) components and/or software programs like rule-based or fuzzy controllers; it can be simple or structured as a hierarchy of main controller and sub-controllers. In each of these cases, the controller has to be differently designed and interfaced to the process being controlled.

The control operation itself may be performed manually or automatically, be either open or closed loop and either work continuously with respect to time or in a discrete fashion, as will be indicated in the following sections. These control concepts and methods are not only applied to process control systems but may be as well applied to psychological and economical systems [Dorf 92]

2.1.1 Open-loop Control

The open-loop control paradigm is both straightforward and normally inexpensive to implement. The process to be controlled is modelled; and then a suitable controller is designed to meet the user's needs, like response time and input/output characteristics. Thus, during the process operation, the user will be able to change the output by altering the command signal applied to the controller. As shown in figure 2.1, a sub-system referred to as the controller is introduced in front of the controlled process. This introduced sub-system is more convenient for the user to apply the input rather than applying it directly to the process.

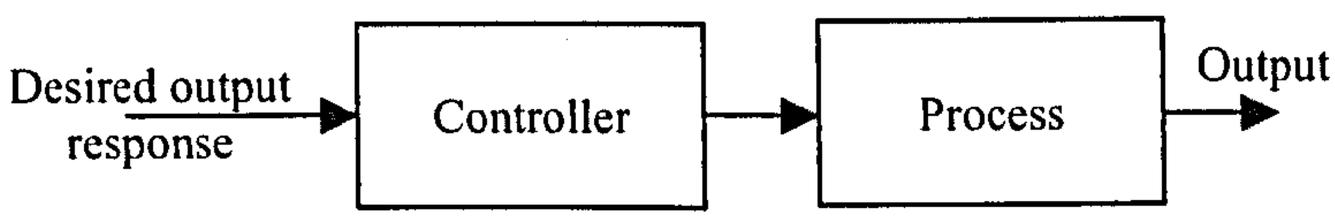


Figure 2.1, Open-Loop Control System

For example, the gas knob in the kitchen cooker is considered as an open-loop control system. The user turns the knob; through some mechanical device the knob position changes the amount of the passed gas that in turn changes the amount of the thermal energy provided by the cooker to cook the food. The knob is considered as a mechanical controller that has the angular position as its input and the gas flow as its output, while the process itself has the gas flow as the input and has the thermal energy as its output. The input-output characteristic is usually calibrated before the user starts controlling the process; after calibration, the process output will be driven towards the desired response as the user applies the corresponding input.

2.1.2 Closed-loop Control

The input/output characteristics within an open-loop control system can vary because of external noise applied to the process or even the aging factors of the physical components. Hence, a need for a dynamic compensation created the concept of closed loop control [Dorf 92]. The idea of the closed-loop utilises a feedback data flow path from the output of the process backward to the controller [Dorf 92]. The feedback concept involves measuring the output, comparing it to the desired value and compensating for the difference between the actual output and the desired one. Although closed-loop control can be considered more complicated and expensive compared to open-loop control, it can reduce the effects of the factors that change the input/output characteristics; thus, closed-loop control can reduce the deviations due to aging factors of the process components, sources of noise, other interference or replacing one of the process components by a slightly different one.

Figure 2.2 shows the building blocks of the closed-loop control system that contains the process to be controlled, the measurement sub-system and the controller. The measurement sub-system contains device(s) to sense the output variations and convert them to the same range as the desired

output response range. Meanwhile, the controller has two input sources: the actual measurement from the process and the user requested command that represents the desired output response. Apart from these two inputs, some controllers have extra parameters that can be adjusted to control speed of

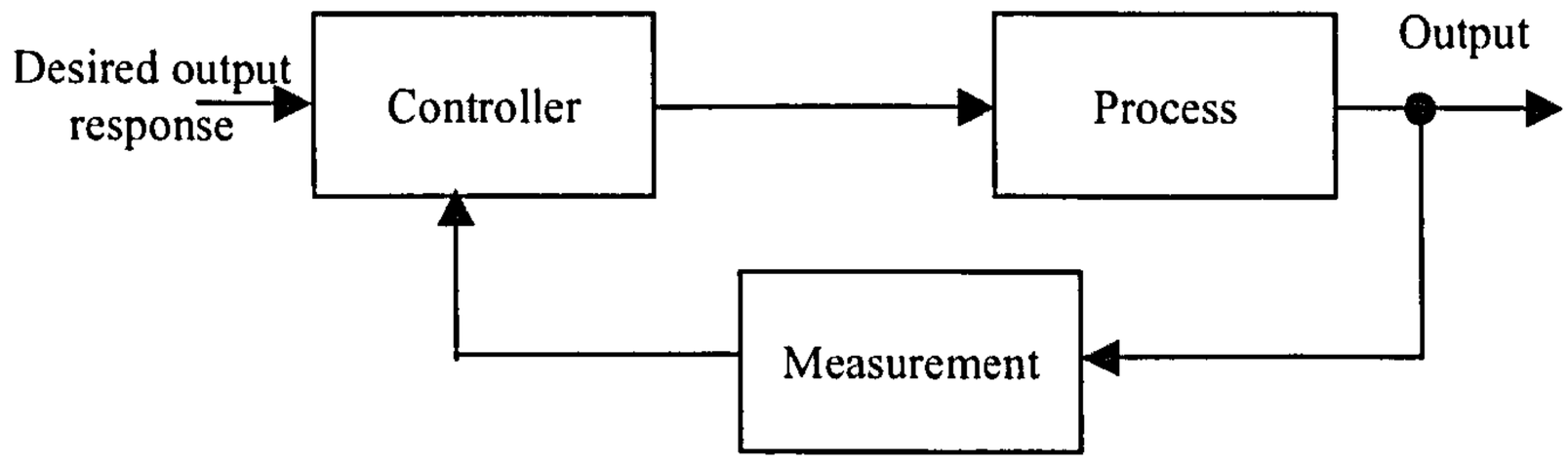


Figure 2.2, Closed-Loop Feedback Control

response or damping behaviour. The output of the process is measured and fed to the controller that applies an appropriate input to the process in order to drive the output to reach the desired response.

The measurement process is of great utility, especially in case of automatically controlled systems. The measurements can provide extra information as follows:

- **Stage Indications.** Systems like production cells have sequences of stages, for example, feeding a blank metal to the press, stamping the blank metal, delivering the stamped metal; such a system needs information so as to know where the blank metal is at any moment and which actuators of the production cell to activate and which to set to a ready or wait state; this can be achieved by providing sensors to detect the existence of the metal inside the press or on one of the belts.
- **Fault Awareness.** Measuring the actuator outputs provides helpful guidance to detect faults and malfunctioning; this is usually achieved by installing an extra sensor attached to each actuator; during the operation, the actual output measured by the sensor to the input applied to the actuator will be compared; in case of a mismatch between the two values, faults can be detected.
- **Alerting.** In some control systems that may manifest hazards or risks to humans, mal-functions should be detected and promptly activate alarms to alert humans around the process site to take the suitable safety precautions. For example, fire-alarm sensors can be devised to measure excessive smoke or heat that indicates the existence of a fire; the controller should be devised to switch the system off and immediately operate sound and light alarms.
- **Commanding.** When it is required to change the process outputs, the user can achieve this either by changing the input(s) applied to the controller that reflect the desired output response, or through some sensors like switches that are directly accessed by the user for purposes like switching on/off or changing the operation mode. Besides these purposes, commanding sensors can be used during process operation to select between different alternative actuators or sensors.

2.1.3 Manual Control

In some closed-loop control systems, the feedback data flow path can be conducted by a human who first measures the process's output and then the actual measured output response is compared to the desired response; finally, this human takes a decision on the appropriate inputs to apply to the process to yield the desired output. Figure 2.3 shows a closed-loop control system to control the tank

head; the human maintains the tank level at a specific value or within a range by changing the valve opening. The human in this case performs the role of measurer and controller while the process has the valve opening as input and the tank level as output.

The level of human interaction with the control system defines whether the system is manually or

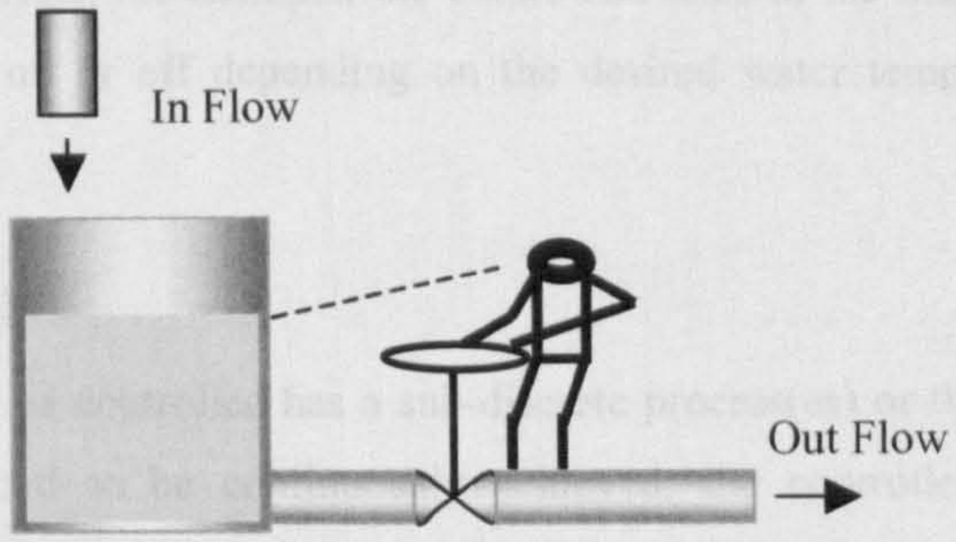


Figure 2.3, Open-Loop Control System

automatically controlled. Deciding between manual and automatic control depends on many issues, such as the following:

- Is it a safe environment for the human to work in?
- Is the human response fast enough to apply inputs to the process, especially in cases where there are multiple inputs and multiple outputs?
- Is the human response precise enough to be accepted?
- Is the automation of the process control system and introducing measurement more expensive?
- Can the human controller detect fault and risk situations and hence shut the system down safely?

2.1.4 Automatic Control

Automatic control is better explained compared to open-loop control and manual control, as it is usually a closed-loop control system in which the measurement and the feedback loop is achieved without human interaction, although the interaction can involve sending commands to the controller and varying the controller parameters.

In order to automate the transfer of measurements to the controller, sensors are employed in the system. These sensors convert physical quantities like pressure, temperature, weight, distance to electrical current or voltage; sensors can be used to measure the different physical quantities; and then by means of signal conditioning, the amplified signals can be fed back to the controller.

2.1.5 Continuous Control

When the process is being controlled continuously over a period of time, the control system is called a continuous system. Taking into consideration that the sensors and actuators as sub-processes require some time to settle on their response, the overall system is dynamic and these characteristics should be considered. There are techniques to model the process, actuators, and the sensors and hence to synthesise a suitable controller to fulfil the user's needs. For continuous control, differential equations are usually used to describe the system; the differential equations can be approximated in linear form; and hence Laplace transforms (S-domain) [Dorf 92] can be obtained for further controller

design and analysis. The controller can be realised by means of electrical elements like operational amplifiers, capacitors and resistors or alternatively using pneumatic components according to the nature of the process nature and the output of the transducers used in the measurement.

In some other cases the required controller characteristics can be non-linear for easier manufacturing or simpler analysis; for example, the controllers used in the indoor heaters are of ON-OFF type that either switch on or off depending on the desired water temperature and the actual measured one.

2.1.6 Discrete Control

When the process to be controlled has a sub-discrete process(es) or the control actions to be performed are too complicated to be continuously achieved, the controller chosen is usually a computer program. These programs have a discrete nature. Discrete Control [Dorf 92] means the output and the input of the sub-system will be considered only at discrete intervals of time; usually these intervals are equal-spaced, as shown in figure 2.4 [Dorf 92].

The sub-systems introduced to achieve discrete control will not differ to a vast extent from the usual closed-loop control; figure 2.5 shows the discrete closed-loop control system where the computer replaces the controller. The process output is measured as before, but the sensed values will be sampled and converted into digital form to be fed to the computer program; following this step, the computer program executes for a single cycle and applies the output to the actuators of the process after changing it into analogue form to drive the desired response.

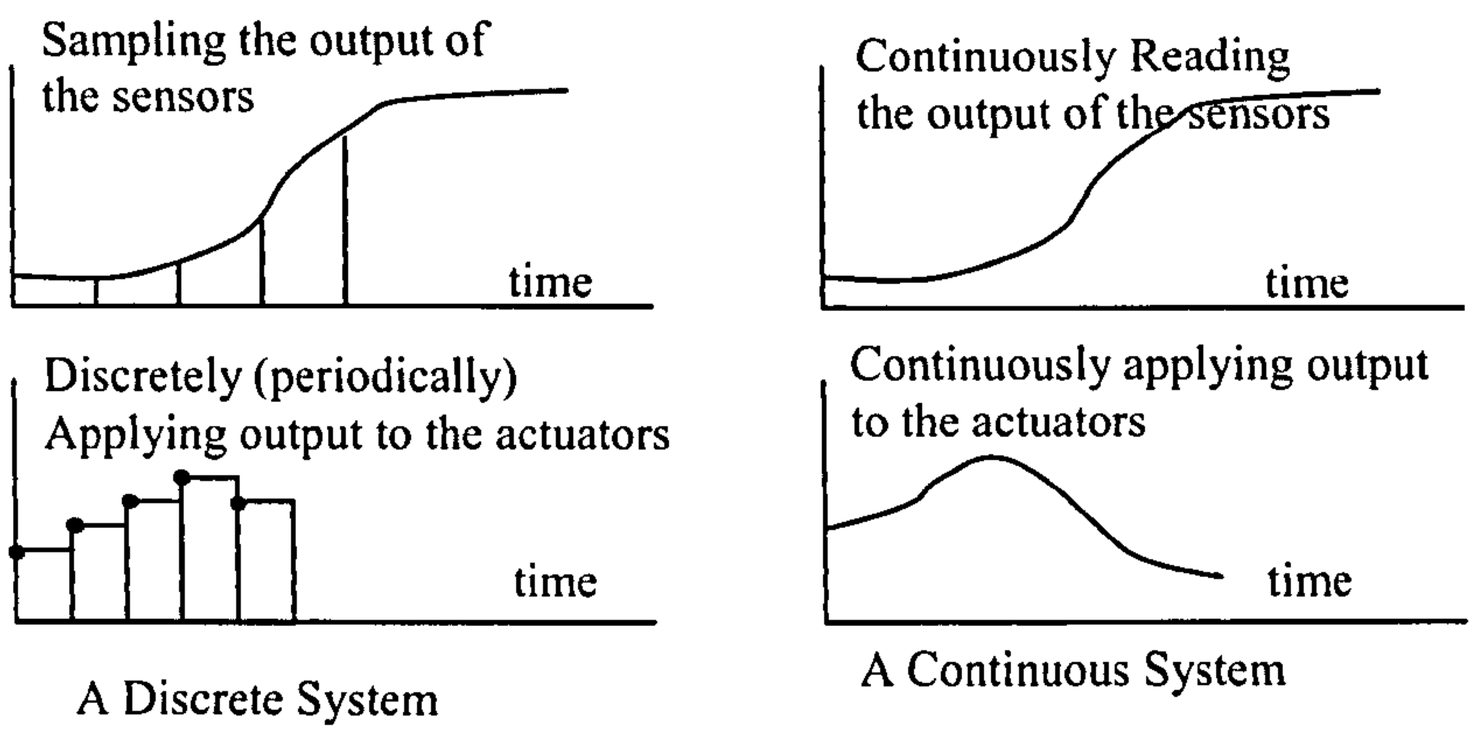


Figure 2.4, discrete vs. continuous systems

The program to control the process can be a software routine that is continuously executed, cycle by cycle; in each cycle, the process outputs are measured and, then, a decision can be made using rule-based expert systems [Winston 92] or a fuzzy logic controller [Mamdani 93]; usually this decision requires time during which the process outputs are sampled and the control actions are applied to the process inputs; then, the cycle repeats again. Alternatively, the program may be hardwired and use a microprocessor to execute its instructions to apply the control action.

Alternatively, for some families of process control systems, the chosen controller is a programmable hardware controller like a PLC (Programmable Logic Controller) [Storey 96], which realises patterns of logical and temporal expressions between the process sensors' and actuators' variables; PLCs have two circuits: control and power. The power circuits are interfaced to the actuators

to apply the computed values from the control circuit in each cycle, while the logic circuit is interfaced to the sensors. In each cycle, each actuator output is computed using logical expressions from the logic circuit and then applied to the corresponding actuator.

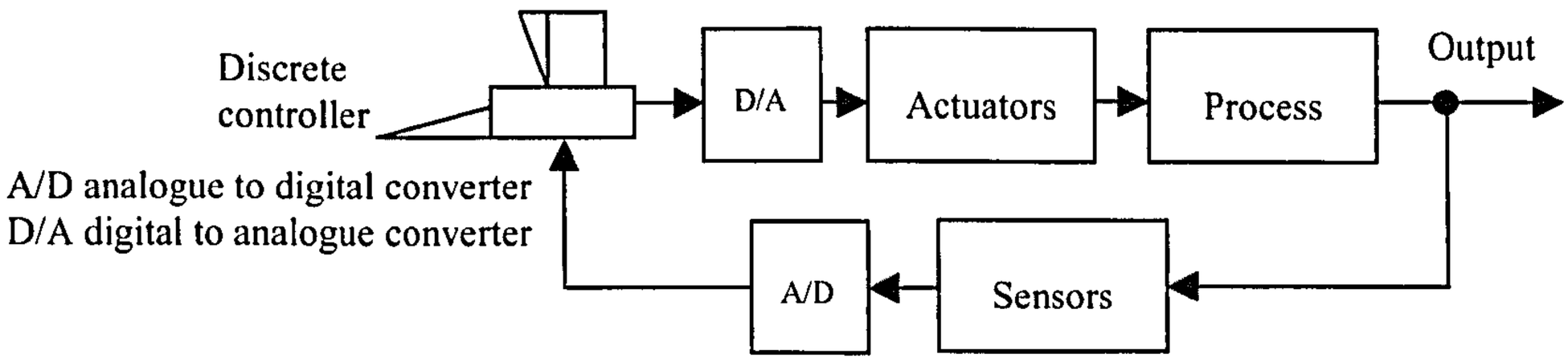


Figure 2.5, a discrete control system using the processing unit as a digital controller

2.2 Research Problem

Process control systems are built of different components; each of these components can be regarded as a separate process that has sensors and actuators. Figure 2.6 [Leveson et al. 94] shows the overall system: the process to be controlled, the sensors, the actuators, and the controller to be built as a software program. The process is usually controlled through applying commands to the controller or directly through sensor readings related to push buttons or switches.

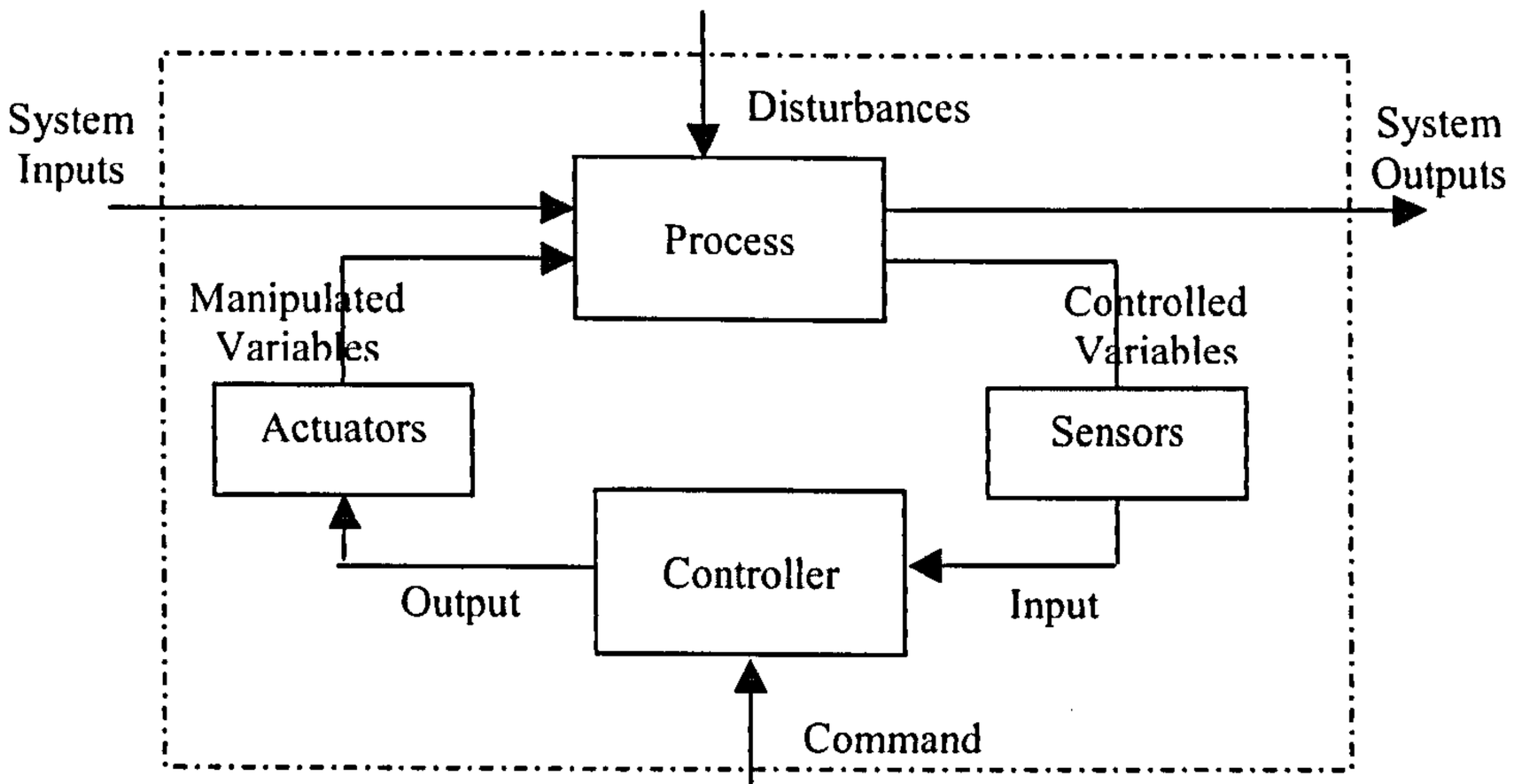


Figure 2.6, A Basic Process Control Model

The controller's main task is to manage the whole system's operation through reading the sensors and then applying the appropriate settings to the actuators. This cycle will be periodically repeated while the system is in operation. Since the number of sensors and actuators can be large in process control systems and the decisions made by the controller can be sophisticated, we will rely on software controllers that operate discretely in periodic cycles; in each of these cycles the new sensor values are recorded, and then, depending on these values and the current state of the controller, decisions are made to alter the actuators' values. Structuring the control operations into modules results in easier debugging and simpler understanding; we will mainly structure the controller modules as a main-controller module and number of actuator-modules.

Like all software applications, the controller requires a development lifecycle so it can be devised to control the process correctly and safely. This lifecycle necessitates different types of expertise to identify both the controller and process interfaces, formulate the input-output

characteristics into cause-effect rules, perform risk and/or hazard analyses, build the outlines of the controller operations, add programmatic refinement, test the developed application, verify and validate the requirements, specifications and implementation at the different levels. Looking at the required expertise, we could conjecture that the co-operation between these experts can be achieved as follows:

- **Process Control Systems Expert.** This person can be the chief systems engineer, with experience of process control families, who can provide general requirements segments that can contribute to different applications. For example, this person can provide details of the most frequently used components like valves, presses or robots. This information can be stored in a softcopy form, such as a library, which can be accessed later to decrease the effort required to construct different process control systems.
- **The Specific Application Expert.** An expert person who is conscious of the operation of the specific process control system, decisions to make when meeting obstacles or facing conflicts, its safety conditions, how to recover from faulty situations, etc. This person is supposed to construct the requirements for the controller; he will be able to use general knowledge about the process control family's construction and basic elements from the library built by the chief system engineer. The validation of the system's operation and safety conditions should be satisfied from the constructed requirements; the systems engineer, by means of a software tool (GOPCSD), will be capable of building the skeleton of the operation hierarchy of the controller software program; the abstract segments of formal requirements or specifications will be refined to implementations and analysed within a formal method environment by a software engineer.
- **Software Expert.** An expert software engineer who knows the fine details of the B formal method and can analyse the generated B specifications that have been generated by translating the requirements provided by the systems engineer. The software engineer will be able to refine the specifications and validate them using the appropriate formal tools provided within the B toolkit.
- **Co-ordination Tool.** A tool will be required to coordinate the transfer of responsibility between the mentioned experts; the coordination should be achieved with as minimum interference as possible to decrease the effort and time and increase the efficiency for each person within his domain; i.e. the chief process control engineer does not have to know details of the particular process control system, as well as the systems engineer not being obligated to know details of the B formal method; and finally the software engineer will be provided with specifications in the form of B machines which are accepted by the systems engineer as operationally satisfactory.

2.2.1 Assumptions about the Expert humans

As explained earlier, the systems, software, and chief systems engineers are involved in creating the controller requirements, specifications and implementations. We list here the assumptions about the knowledge about process control systems the software and the systems engineer may, should and do not have to know.

2.2.1.1 What the Systems Engineer should know

The systems engineer has to know the system's basic construction (the various components that make up the system and their input/output characteristics); the system operation modes and the appropriate conditions for each of them; safety and hazard situations and how to shut down the system safely; how to detect existing faults, whether or not to continue system operation or partially or completely shut down the system; how to formalise the textual requirements into conditional assignments; how to debug the requirements from the operational point of view; how to judge the performance of the formally specified controller and possibly choose an alternative design to implement.

2.2.1.2 What the Systems Engineer may know

The systems engineer may know general information about process control systems' components, modes of operation, integration, and interfacing; he/she may have knowledge about programming languages, and what can be achievable through the use of computing processors.

2.2.1.3 What the Systems Engineer does not need to know

The systems engineer does not have to know either about the details of the B specifications, programming-oriented refinements of the system's operations or the structuring of the generated controllers into sub-modules.

2.2.1.4 What the Software Engineer should know

The software engineer should be aware of the application variables and the data types assigned to each of them, how to refine the different operations starting from the pre- and post-conditions specified by the systems engineer, how to carry out proofs within the B toolkit, how to debug the specifications from the programming view, how to refine the specifications by adding more details to it, and finally, how to implement the specifications into a high-level programming language.

2.2.1.5 What the Software Engineer may know

The software engineer may have knowledge about process control systems and specific knowledge about the application components and operations.

2.2.1.6 What the Software Engineer does not need to know

The software engineer does not have to know about the detailed operations of the application or any internal detail about component input/output characteristics.

2.2.2 Problem statement

We can summarise our discussion of the differing roles and expertise of the systems and software engineers as follows:

In process control systems, where the control systems are built of existing (physical) components, it is difficult to separate the concerns between the software and process control engineers in developing the software controller. These difficulties usually result in delaying the requirements enhancement and

error discovery to a level where the requirements will be refined and formalised using a particular implementation or design form/format/language. Moreover, these requirements formalisations and refinement decisions will be assigned to a software engineer who is capable of using the implementation or design environment, not by the systems engineer. Thus, the chances of developing unsatisfactory software applications will be higher.

Considering the previous assumptions about the expert humans involved in creating process systems controllers, we can propose the following solution:

Creating a development method based on the goal driven method of KAOS (will be explained in detail in chapter 3). This method should reduce the interference of concerns between the systems and software engineers, as well as systemically guide the systems engineer to refine and formalise the very abstract and informal requirements. Hence the systems engineer can be in a better situation to correct the requirements bugs and enhance the requirements. To continue the development, an automatic translation to a detailed B formal specification (The decision to use the B formal method will be explained in detail in chapter 4.) would be required so that a software engineer can be in charge after insuring the user needs have been formalised and refined correctly.

To be able to assess the method through examining process control case studies, it was a necessity a have a tool that automated the method steps. This tool's inputs, outputs, users and assumptions are listed in table 2.1.

Table 2.1, description of the supporting tool

Input	• Structural descriptions of the application • Separate parts of the application requirements, like components' requirements and the applications' high-level, abstract requirements.
Actor	• Systems Engineer
Tool	• An interactive software requirements analysis tool (the GOPCSD tool)
External Data source	• Commonly used components within process control systems • High-level requirements templates for process control systems
Output	• Structured formal requirements in the form of a set of alternative goal-model(s) (an intermediate output) • B machines as formal specifications for the controller of the process application (the main output)
Assumptions	• It is more straightforward to separately check the requirements rather than to only debug the applications after a single combined stage of requirements and specifications. • The software engineer will be able to refine and process the generated formal specifications without a need to refer back to the systems engineer regarding the application's operational characteristics. • The systems engineer will be provided with enough support and guidance to understand and construct the hierarchy of goals.

This approach is considered a novel and useful one because it specialises a general concept of top-down refinement to the process control field where more specific refinement patterns can be identified. It is also useful because it encapsulates relevant software engineering concepts within a systematic engineering approach that can lead to achieving successful design amongst the conventional systems engineer community. In addition, it attempts to guide the transition from the informal to the formal description, which is conventionally solved in a way that is biased towards the formal from the software engineering perspective and towards the informal side from the systems engineer's perspective.

2.2.3 Research Boundaries

In this section, we fix some boundaries and specialisations for both the research and the developed tool in order to maximise the benefits of an early development stage like requirements gathering. Thus, we set some boundaries to limit the research as follows:

- The systems engineer is assumed to be capable of identifying the physical components of the application and their inter-relationships. Thus, the development phase within the GOPCSD tool can start by creating the agent, component, and variable lists without explicitly entailing constructing ERD or class diagrams as in KAOS.
- The main variable type supported by the tool is the enumerated (user defined) type. The variables of different types can be represented by mapping their value range to an enumerated range, or alternatively using refinement machines within the B toolkit environment.
- The evaluation of the alternative solutions will be a responsibility of the systems engineer; however, the tool can generate the different solutions and check their consistency and completeness, the systems engineer has to select the convenient solution.
- The allowed patterns of refinement of the goals will be hardwired within the tool. Thus, it will not be possible to directly use different patterns of refinement.
- The tool will not address any further programming refinement beyond generating the B machines' specifications; this prepares the stage for the software engineer to be in charge for the remaining steps to generate an implementation.
- The generated specifications in the form of B machines will be documented with informal description segments of the requirements goals as comments, which are assumed adequate to process the specifications and build an implementation and to trace the specifications to the requirements without necessitating guidance or suggestions from the systems engineer.

2.3 Other Approaches

There are other approaches to gather the requirements of process control systems and/or to transform the requirements into formal specifications formulated in languages such as Z, B and other formal specification formalisms. However, some of the approaches assume that the user can supply the method with complete formal requirements, while others assume the user of the method should have

miscellaneous knowledge of both the sophisticated logical and mathematical details and the sequences and cause-effect operations of the process control applications. On the other hand, other methods start much earlier than expected in a well defined domain, especially within a family of applications like process control systems where there should be some minimum understanding level of the relationships between components within the application. In the following sub-sections, we address the relevant related work. Our research attempts to combine the constructive features from each of the following methods with the goal-oriented features of our extension of KAOS. (The reason for choosing the goal-drievn KAOS method will be expalined in detail in chapter 3.)

2.3.1 Graphical Design of Statecharts

In [Sekerinski 98], Sekerinski builds B AMN specifications starting from state charts for reactive systems. A state chart is built for the entire reactive system as a conjunction of a number of state charts for each of the reactive system's components. The transitions between the different states are driven by the events produced by sensor readings, while the expected actions from the reactive system, like changing actuator values, will be emitted during the appropriate transitions. After designing the state charts for the reactive system, Sekerinski translates the component state spaces into variables in B AMN, the possible states for each component into different domain-dependant enumerated sets that the variable of the corresponding space will be assigned to, and finally the transitions into B AMN operations that group together all of the state transitions driven by the same event. For those state transitions that require emitting actions, the actions will be emitted in parallel to the change of the state space variable's value (assigning the new state to the state variable).

The process of translation provided in this method is very clear and encourages the use of a state chart as an initial draft for designing the reactive system. However, the method itself may be embedded within a user interaction interface, where the systems engineer can easily construct the requirements and validate them. The graphical method assumes the transitions and the composite and parallel states are easy and straightforward for the user; still, it is more common to expect the systems engineer to specify what the reactive system should do rather than to be able to build a statechart model.

2.3.2 Reactive Systems Development Support tool (RSDS)

In [Lano et al. Y2Ka], Lano et al implemented a graphical design method for reactive systems, RSDS (Reactive System Development Support) for generating specifications in VDM [Jones 90] and B, and implementations in JAVA. The RSDS tool assumes that the user has full awareness of the component-structure of the application and the detailed relationships between the different invariants governing the application. Thus, the user builds the application through defining context diagrams built of sensors and actuators representing the real application components; then, the controllers can be added to the diagram and assigned sensors to read and actuators to control; the user is assumed to be involved in such assignments, which are accomplished by a drag and drop style. Following this step, the user specifies invariants that govern the input and output variables, as well as any restrictions and constraints. The tool checks completeness and consistency of the invariants; furthermore, it has the ability to structure the generated B controllers in different ways. Through

invariant pattern identification, Lano et al structure the B controllers as indicated in [Lano et al. Y2Kb] (as will be illustrated later in chapter 4).

On the one hand, RSDS hides the details of the formal methods as well as the Java programming language from its user; the user will be guided to resolve any inconsistency that may exist in his formal requirement, in addition to completing any missing case (combination of the sensor values). On the other hand, the tool assumes the user provides correct requirements as separate formal invariants. Besides, the user gets involved in controller assignments, defining internal states of the actuators and sensors, and compound logical expressions of the invariants. Thus, the user will not be able to reason about the requirements, separately, nor analyse the relationships between the different invariants. The tool does not allow an earlier chance for the user to validate the requirements before testing the Java programs or the B specifications; at this time, the bugs may be due to either the requirements or specification stages, complicating the process of eliminating the bugs. From RSDS SMV code corresponding to the requirements specification can be manually translated to check the preservation of the temporal properties of the reactive systems.

2.3.3 Requirement Specifications for Process-Control Systems

Leveson et al. in [Leveson et al. 94], developed an approach to build a single state-based model (Requirements State Machine, RSM) to describe process-control applications; such a model includes all the information required to describe the "blackbox" behaviour of the inner components of the system. The RSM model was developed first to use other existing specification languages, but later, the research team formulated RSML (Requirements State Machine Language). The approach has the ability to model the behaviour of the components of the process and the controller itself as well as the communication between them. As an example given in [Leveson et al. 94], figure 2.7 indicates the unified model for a single component which has input and output variables and four parallel states (X, Y, Z and W).

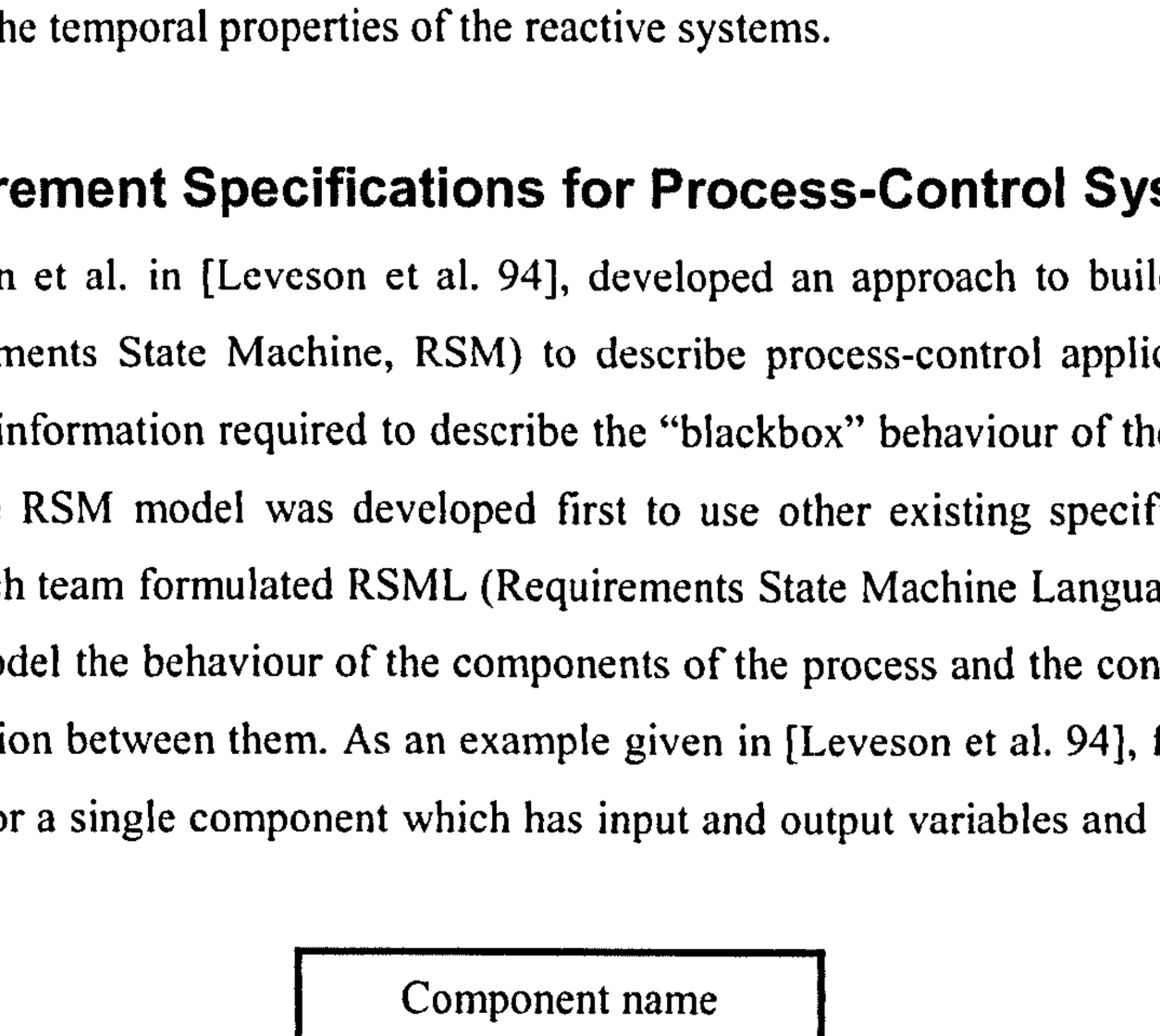


Figure 2.7, representing each component by a state machine with associated variables in RSL

The component input/output characteristics are described by means of the internal states; for example, when the component has the parallel states (X1, Y2, Z1 and W2), the variable1/variable2 characteristic differs from being in state (X2, Y3, Z2 and W1).

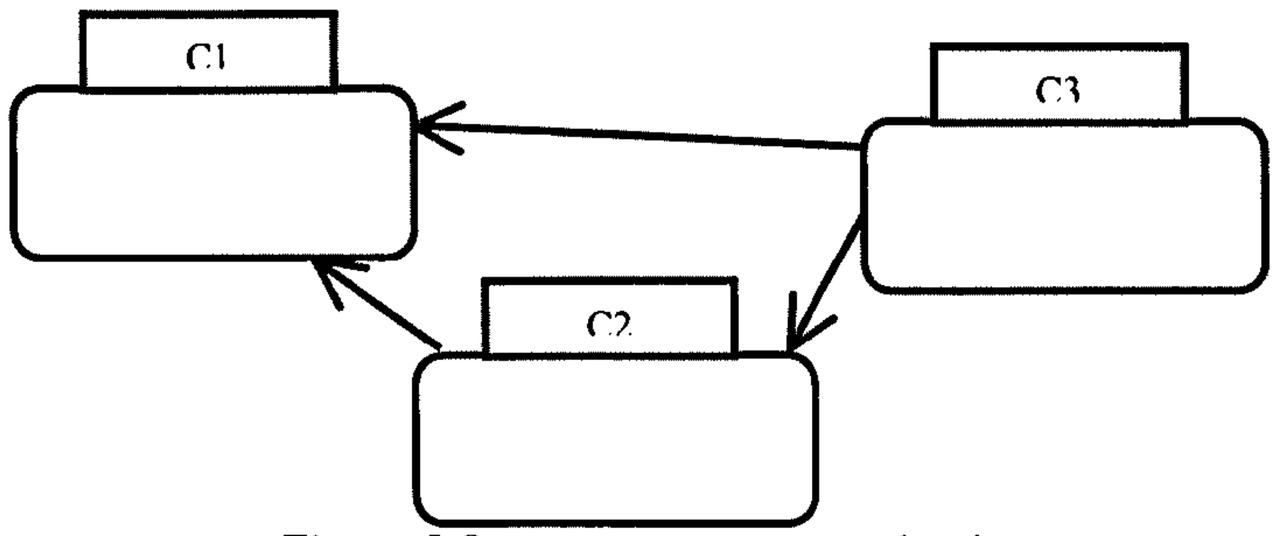


Figure 2.8, component communications

The internal state can be changed by two different types of events: internal events that happen within the components and cannot be passed from one component to another, and the external or the inter-component events, which stimulate the components to change their internal states.

In figure 2.7 [Leveson et al. 94], the arrows from one state to another indicate the internal transitions, while the small arrows, with one attached to each parallel state, indicate the initial state; however, in state X the small arrow points at a transition branching point (C), where each outgoing arrow has a disjoint condition, to start at an appropriate state corresponding to the condition associated with the arrow. The transition conditions are usually written in predicate calculus.

An important feature of RSM is that the effort required to specify and construct the requirements of the system will be broken down into smaller efforts for each component, as shown in figure 2.8 [Leveson et al. 94], to give the opportunity for the specifications to be focused separately on each component, then to the communication between the components, which will be easily managed by setting the appropriate conditions and the input variables for each component. The "black box" model separates the specification of the requirements from design, simplifying the model and making the requirements model easier to construct, review and formally analyse.

Although there is a strong focus on component-based development, two main drawbacks can be noticed: the controlling activity is not centralised, but distributed among the different components and the RSML language is still required to be translated into a high-level language or standard formal specification language, such as VDM, Z and B. Otherwise, a well defined translation will be required to generate a formal specification correspondingto the RSML requirements.

2.3.4 Four Variable Models and the SCR Method

In [Parnas and Madey 95], Parnas and Madey have presented a model for early specification stages for control systems. In this model, the application variables are categorised into four types: input, monitored, controlled, and output. The mapping between the different variable types reveals an understanding of the control functions of the system, which differs somewhat from the very idea of software requirements, which can hide some of the control application details. These details can be considered as refined specifications from the software view, whereas they are essential from the control view. One of the methods that adopted this model is the SCR (Software Cost Reduction) method [Heitmeyer and McLean 83]. In SCR, the abstract requirements of safety-critical and high

assurance systems are formalised using a formal specification language based on the four variable model. SEVERAL TOOLS (SCR toolset [Jefford and Heitmeyer 98]) have been developed to apply the SCR method [Heitmeyer et al. 98], each tool of the tool set can detect some type of the user errors like ambiguity and inconsistency.

Although the four-variable model is suitable for specifying control systems, it assumes the user has a correct initial expectation of the control functions. However, this might not be the case, especially for large systems, where the user can start development with rough and informal understanding of the application and, then, with the help/guidance of the development environment develop a refined version of the specification.

In [Landtsheer et al. 03], an effort was focused on translating goal-models (more description is provided about goal-model in chapter 3), which have a mixed formal and informal nature, to a formal SCR format. Although this effort bridged the gap between the early requirements and the specification analysis stages, the result of the formal checks of the requirements decisions are related to the SCR formal specification and not traced directly to the requirements goals, which ends up with the requirements as a dependant stage that will not provide a complete justification for the requirements decisions of its user.

2.4 Conclusions

Although process controllers vary dramatically, they share the same control concepts. Sensors are employed in process control systems to automate the closed-loop control and to achieve different responsibilities like commanding, alarming and fault detection. The research focuses on closed-loop discrete control of systems requiring the building of software controllers.

Some assumptions have been made about the process-control systems engineer who uses the tool to construct the specifications for the process control applications, as well as about the software engineer who processes the generated specifications (B machines) within the B toolkit environment.

Some boundaries have been fixed for the research in order that it can be more readily achieved. Finally, we briefly mentioned other related approaches to generate formal specifications for process control systems and reactive systems in general.

Requirements Engineering 3

In this chapter, we introduce the requirements stage and emphasize its importance as the first stage within the lifecycle of developing software applications. Then, we focus on software requirements engineering and describe the goal-oriented requirements analysis method of KAOS as one method to structure and check software requirements. Finally, we review other goal-oriented requirements methods.

3.1 Introduction

IEEE standard 729 defines a requirement as *"a condition or capability needed by the user to solve a problem or achieve an objective; or, a condition or capability that must be met or processed by a system to satisfy a contract, standard, specification, or other formally imposed document"*[1] [IEEE 84]. Gathering requirements and carrying out analysis studies are usually the first steps before constructing any engineering systems. The requirements analysis deals generally with the client needs and perspective: what the system is supposed to do and which constraints the system should obey to conform with the environment where it is supposed to operate. Some other analysis studies should be carried out, like safety, HAZOP [Storey 96], cost studies, environmental studies and other specific analyses according to the nature of the system to be constructed.

The requirements analysis stage is crucial to clarify what are the required system's functions, what is the upper bound for cost and what will be the estimated effort and time for developing it. Furthermore, some early measurements, like response speed when changing the input commands, can be investigated at this early stage. All these clarifications are essential to refer to, especially in the case of contracting a second party to construct the system for the client. In addition to the clarification aspect, this stage is important to capture the requirements bugs,

[1] We use italic style to indicate quoted text

inconsistencies, incompleteness, or ambiguity within user needs that can result in erroneous systems or hazards that cannot be overcome at later stages without higher cost and effort.

As an engineering activity, the requirements gathering process involves using standard and documented methods, like using standard diagrams to model the requirements and the local environment where the system is intended to operate. Some other engineering methods can be used, like reusability and a hybrid between different accepted methods for gathering the requirements.

3.1.1 Software Requirements

Within software applications, the requirements gathering and checking stage is considered as the first stage in the application development lifecycle; a complete understanding of software requirements is essential to the success of a software development effort. Requirements and software specification may appear to be relatively simple tasks, which may consequently have less attention paid to them. But this is not the case because chances for mistakes or misinformation abound and, without early checks and corrections, these mistakes will remain hidden, and affect the final application. *Requirements analysis enables the systems engineer to specify software function and performance, indicate the software's interface with other system elements, and establish design constraints that the software must meet* [Pressman 92].

The branch of software engineering called Requirements Engineering (RE) is concerned with the real-world goals for software systems as well as their constraints. It also involves precisely specifying the software's behaviour, and enabling software evolution over time and across software families [Zave 97]. On the other hand, Jackson, in [Jackson 01], states that the requirements for developing software programs are the set of rules defining the software program to be developed and those of the environment where the program will work and interact. Thus, the software specification can be considered as a subset of the requirements, because they only describe what the software program should do. Jackson used a basic definition of recurring problems to decompose all software development to a set of such problem frames.

The operations of the software application can be abstracted, tested and validated before implementing the final application. In particular, the software application in general can be specified formally, for example, as pre- and post-conditions for each segment of the application. This abstraction hides the details of how to achieve the post-conditions for each segment starting from its pre-conditions. This abstraction provides a chance to formulate the requirements formally and then be able to check, validate, and animate them in order to have agreement about the requirements, and furthermore eliminate the requirements errors.

No matter how well designed or well coded, a poorly analysed and specified program will disappoint the user and bring grief to the developer [Pressman 92]. *Because the primary indicator of the success of a software application is how much it conforms to the purpose for which it was intended* [Nuseibeh and Easterkrook, Y2K]. In [Davis 03], the requirements phase is considered complete only after the requirements have been specified using normal language/formulae and logic expressions/modelling and analysis diagrams such as ERD (entity relationship diagrams),

DFCD (data flow control diagrams) and STD (state transition diagrams) [Wieringa 01]. These software requirements specifications (SRS) are considered as detailed refinements of user needs. As in [Davis 93], the SRS are considered important because: there are many requirements errors hidden within the developed applications. However, these errors can be detected at later stages of development, but with higher cost for development. These hypotheses are supported by experience in the software industry. This may explain the effort paid to the testing stages especially when the requirements specifications did not have an adequate amount of care devoted to their construction. Furthermore, in [Davis 93], Davis stated that:

> *propagating errors within the requirements phases and defects within the SRS have a direct impact, as follows:*
>
> - *The resulting software may not satisfy users' real needs.*
> - *Multiple interpretations of the requirements may cause disagreement between customers and developers, wasting time and money and perhaps resulting in lawsuits.*
> - *It may be impossible to thoroughly test that the software meets its intended requirements.*
> - *Both time and money may be wasted building unwanted applications.*
> - *Although the user may accept applications that have requirements errors, the applications can still exhibit unexpected and undesired features that can cause hazards, increase the running cost, or have different behaviour from user needs.*

Because of the informal nature of the SRS, it can be stated in different ways and different styles; the SRS should support reasoning about how the application interacts with the environment at the boundaries between the system and its environment and avoid describing any details of how to achieve the application itself.

To assess and evaluate the produced SRS, [Davis 93] suggested some attributes to be measured; although these attributes address different orthogonal or dependent features, they can still provide guidance about how well written the SRS is. A well-written SRS should be Correct, Unambiguous, Complete, Verifiable, Understandable by the customer, Modifiable, Traced, Traceable, Design independent, Annotated, Concise and Organised. It is important to note that, when improving one of these attributes, it might result in less satisfaction of the others; for example, increasing the understandability of the SRS may conflict with completing it.

In conclusion, requirements gathering is a process of discovery, refinement, modelling and specification. In order to have an effective requirements gathering and checking stage, the following aspects should be considered:

- The SRS should be as well-written and understandable as possible.
- The requirements of the new applications should not conflict with the existing applications; this means it is essential to consider the existing related applications as part of the environment as well.

- The requirements should be flexible enough to provide flexibility for the implementation level and allow implementing new, future, related applications that may be developed later without exhibiting conflicting behaviours.

3.1.2 Requirements Reusability

Software reuse, when properly achieved, reduces time and effort required in analysing and maintaining software applications. Moreover, the chances of having poorly formulated applications will decrease because the reused software has been already tested. All of these properties of the development process are considered as important goals that software reuse can help satisfy. It is preferable to be able to reuse requirement pieces and methodologies that have proved to work correctly in similar applications; this trend encourages both using standard methodologies and models for the requirements gathering and structuring the requirements models to be able to reuse parts of it later in similar applications. Requirements refer to specific domains and to specific tasks. Requirements within a similar domain and for a similar task are more likely to be similar. However, techniques for retrieving, updating and consolidating reusable requirements have received relatively less attention than the work on software reuse at the implementation level. This may be related to the large number of methods and tools used for supporting the requirements and specification stages compared to the fewer high-level programming languages used at the implementation stages.

Some efforts have been focused on reusing requirements and specifications segments; these range across problem understanding, analysis, requirements, specification and early design [Leach 97 p6]. Thus, software reuse should be considered as applicable in all of these development stages, as follows: Jackson, in [Jackson 01], suggests reusing problem patterns to understand and decompose complex user needs. In [Massonet et al. 97], Massonet and others use query generalisation to check the analogy between similar systems and then use formal rules to elaborate the requirements for the derived systems. Reubenstein and Waters in [Reubenstein and Waters 91] use specialisation of existing systems requirements to create new systems. At specification level in [Maiden and Sutcliffe 92], Maiden and Sutcliffe utilise analogy between similar systems to import specification segments.

3.1.3 Deciding on How to Gather the Requirements

Since the early eighties, different institutes and organisations have established standards to represent and document the requirements specifications, such as the Department of Defence DI-MCCR-80025A [DOD 88], Institute of Electrical and Electronic Engineering (IEEE) and the American National Standards Institute IEEE/ANSI 830-1984 [IEEE 84], and Naval Research Laboratory NRL A-7E OFP SRS [Basili 81, Heninger 80].

These standards address the organisation of the requirements, but the format or the template used to represent the requirements (formal or informal/functional or non-functional) varies and requires a careful analysis to choose a requirements model that fits the intended application. In particular, the decision to choose one requirements model to use and how to use it

(systemising steps of building/checking the model) needs careful judgement based on precision of representation, understandably, usability, traceability, etc. This judgement strongly depends on the nature of the applications and the nature of the environment where this application is supposed to operate. Not only which model or method to use to achieve better results, but also the automation aspect should be considered; some methods/models are more convenient to be manually used than others, which may be more conveniently automated. For example, developing a database application encourages the use of the entity-relationship diagram (ERD) and the data control flow diagram (DCFD), while more attention should be paid to where this database application will work (for example, a distributed database application's requirements needs may differ from a single-user's needs). As indicated in [Davis 93], the SRSs can be grouped by the external stimulus, system features, system response, class of users, and/or class of functions. In large applications, the SRSs can be grouped using a hierarchy of grouping criteria.

In order to be able to remove requirements misunderstandings, formal models such as Decision Tables and Trees [Moret 82, Chvalovsky 83], Statecharts [Harel 87], Finite State Machines (Mealy, Moore), and-or trees/goal-trees [van Lamsweerde 91], R-nets [Alford 76], and Petri nets [Petri 62, Peterson 77], Specification and Description Language (SDL) [Rockstrom 82] can be used. These models vary considerably in their ease of understanding, modifiability, tracing of user needs, and traceableability at the design level.

A goal-model, as an SRS model, is considered as an understandable, traceable, organised, structured and modifiable, as will be explained in the case studies (chapters 7 and 8). Hence, having considered the aspects mentioned in section 3.1.1 and paid significant attention to reusability and high assurance [Letier and van Lamsweerde 02a], we decided on modelling the SRS in terms of goal-models. In particular, we automate and adapt a version of the goal driven requirements analysis method of KAOS [van Lamsweerde et al. 91] to structure and check the requirements for process control systems.

3.2 Introduction to KAOS [van Lamsweerde et al. 91]

The KAOS (Knowledge Acquisition in Automated Specification) method [van Lamsweerde et al. 91] is a general method to structure the requirements of software applications as well as those systems that include other sub-systems in the form of hardware devices or humans.

KAOS is a Goal-oriented methodology that fully describes the high-level and low-level goals required from a specific system, as well as the objects, and operations to be assigned to various agents within the application. The KAOS methodology provides a specification language, an elaboration method, and tool support [Dardenne et al. 93, van Lamsweerde et al. 95, Darimont et al. 96, Darimont et al. 98]. As will be explained in the following sections, most of the attributes of well-written SRSs are supported in the KAOS method, which motivated us to use it.

3.2.1 Modelling levels

The KAOS approach to requirement engineering provides three levels of modelling: meta, domain and instance levels. The KAOS language is defined through a conceptual meta-model. This meta-

model provides domain independent abstractions in terms of which domain-specific concepts are acquired. The meta-model is composed of Meta concepts (goal, agent, relationship, etc); Meta relationships (ISA, Refinement, Responsibility, etc); and Meta attributes of the Meta concepts and the Meta relationships. The concepts of the Meta level are related to each other through the Meta relationships as represented in figure 3.1. The domain level contains goals, agents, and relationships of the specific application [Letier 01]. For example, in a gas burner system such a domain model is composed of a flame detector and a burner. The domain level components are instances of the Meta level components as shown in figure 3.1 and indicated by dotted arrows pointing from a domain level component to a component within the Meta level. The instance level is composed of objects from different entities having relationships among them. These entities and relationships belong to the specific system. For instance, in a specific burner system, the instance model contains one object like valve1 that is an instance of the Gas valve entity shown in the domain level; the relationships between the entities of the domain level should be preserved in any instance level.

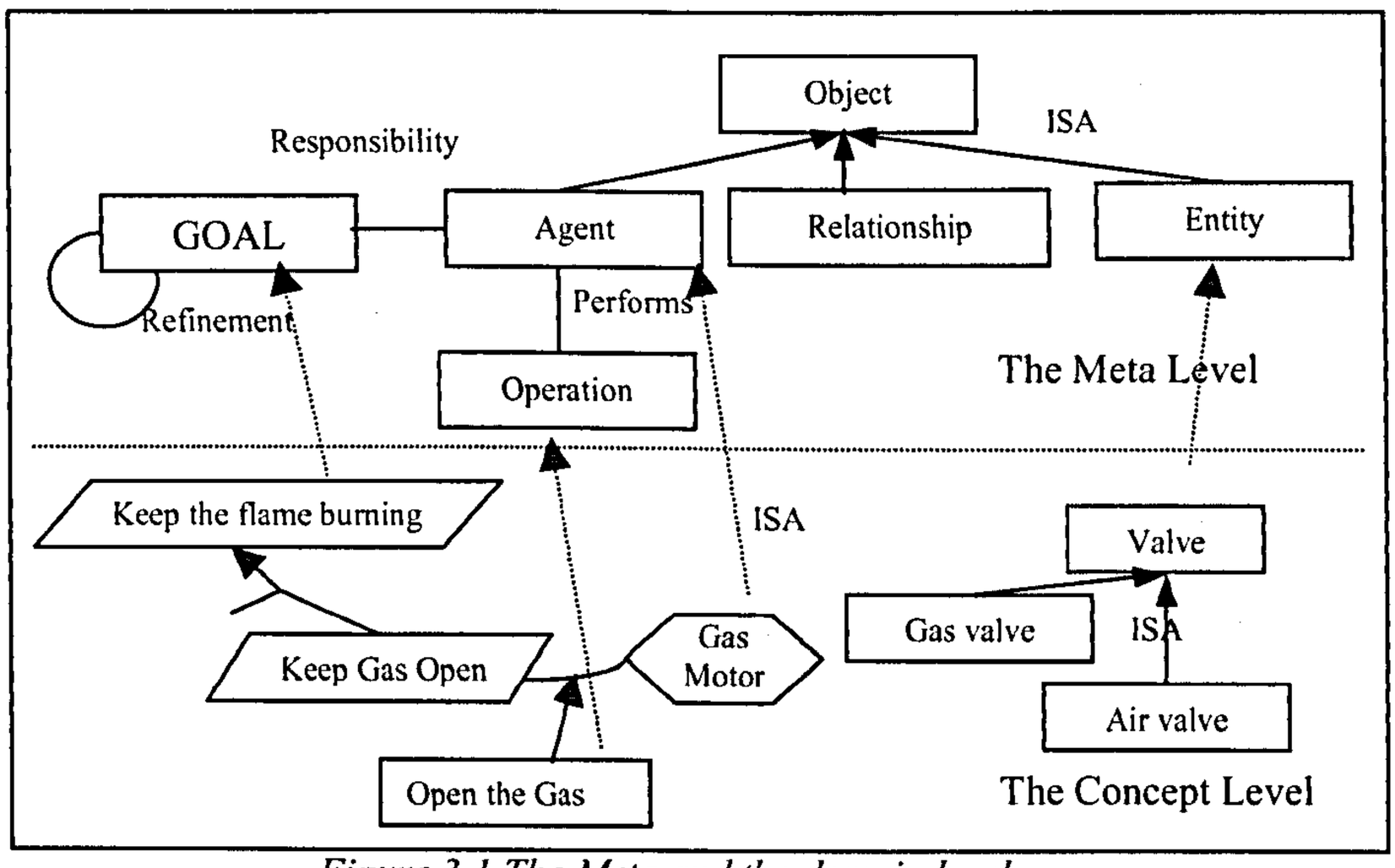


Figure 3.1,The Meta and the domain levels

In figure 3.1 [Letier 01], the Meta level contains six main entities: Goal, Agent, Operation, Object, Relationship, and Entity. The Goal entity has a recursive relationship that represents the sub-super relationship that is to be used to construct the goal-model hierarchy. The agent entity has a direct relationship with the Goal entity indicating that the terminal goals (some of the goals) will be assigned to certain agents. These entities compose the data models of the KAOS method as shown in the following section.

3.2.2 The KAOS Data Models

The KAOS method has four different models: object-model, goal-model, agent responsibility-model, and operation-model. By completing these four models, a full specification for the system operation will be constructed; the object-model identifies the basic objects of the local environment and the system in addition to the relationships between them.

The complete goal-model contains a hierarchy of goals that constitute the main objectives of the system; the goal-model starts at its highest level with the main goal of the system, which is refined to sub-goals until reaching the level of terminal goals, which are assigned to agents to achieve them. The details of these agents and which operations they can perform are stored within the agent-model. Finally, what each agent should perform to achieve terminal goals will be stored as operations in the operation-model. Figure 3.2 shows graphical representations for the different data models.

3.2.2.1 Object model

The Object-model identifies the objects of interest at the domain level. For example, in the gas burner system, the objects are things like burner, flame detector, and flame disappeared. An object can be an entity, relationship or event. The objects can have ISA (is an instance of) relationships among each other to indicate inheritance.

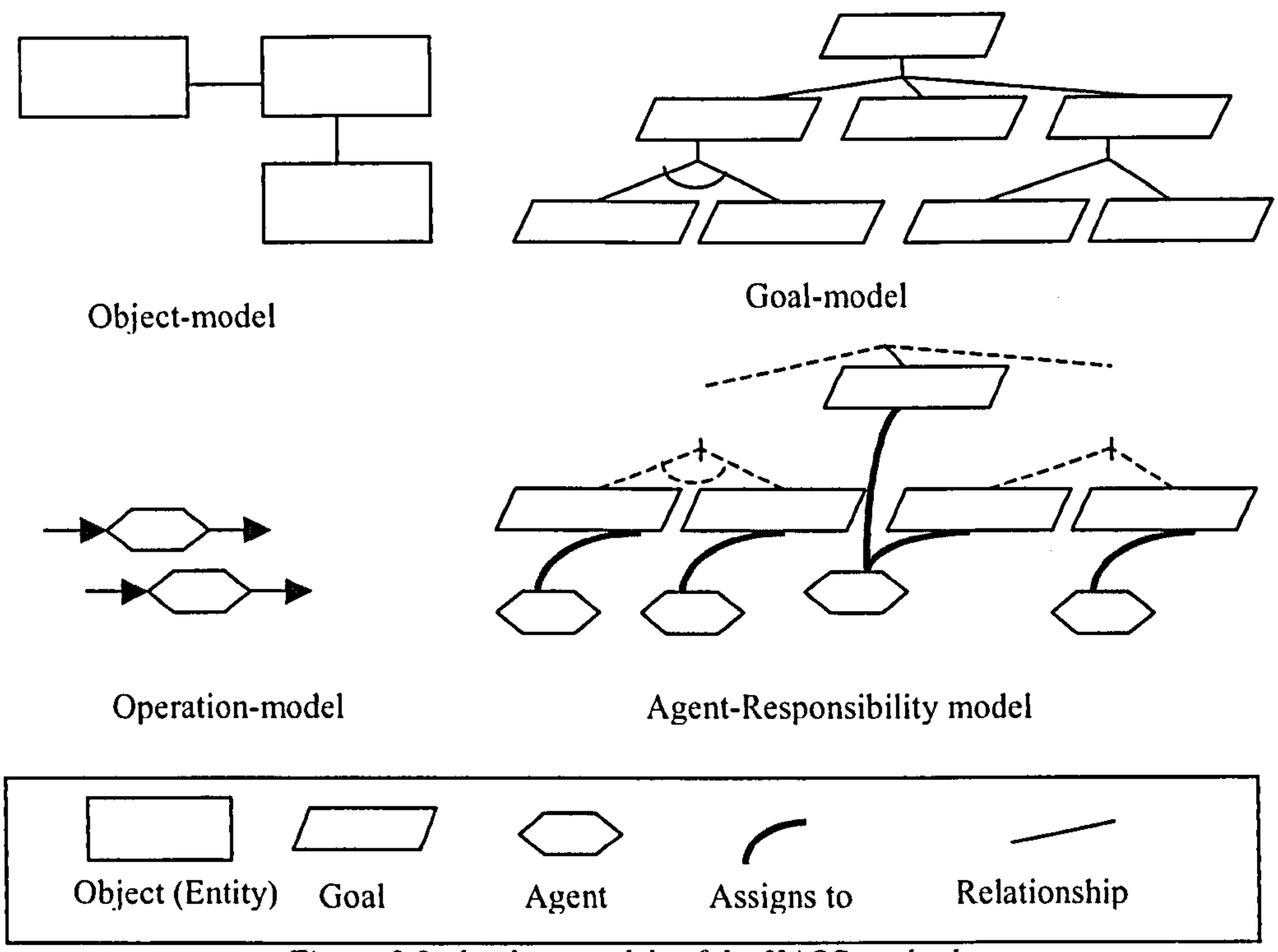


Figure 3.2, the data-models of the KAOS method

3.2.2.2 Goal model

The Goal-model is a hierarchy of goals; each goal models a piece of the requirements, either a functional goal or a non-functional goal such as safety, security, cost. The goals can be simple enough to be terminal goals and assigned to an agent to achieve them, or need refinement to break down the intended task defined by the goal into sub-goals that share some logical relationships among each other (disjunction or conjunction). For each goal, information about the formal and informal descriptions of the goal, sub-goals, and super-goal is provided.

3.2.2.3 Agent responsibility model

The agent responsibility-model declares the responsibility assignment of terminal goals to agents. An agent can be an external device, a human in the local environment, or a software component that exists or is to be developed. The decision of the goal being terminal or not depends mainly on the agent, as the agent needs full observability for the goal's variables and full controllability of the variables changed by the goal. In such a case, the goal is said to be terminal and assigned to the agent.

3.2.2.4 Operation Model

The Operation-model represents the building blocks of what the system should realise. These operations and the other data models of the requirements in KAOS can be used later to formulate formal specifications for the application. The different operations have pre- and post-conditions and can be triggered by specific events sensed by the system. The agent should be capable of performing one or more operations, but each operation is assigned to exactly one agent.

3.2.3 Language Constructs

The KAOS language has constructs to describe the different concepts like goals, agents, operations, and relationships. Each construct in KAOS has a two-level generic structure: an outer semantic net layer for declaring a concept, its attributes and various links to the other concepts and an inner formal assertion layer for formally defining the concept. The outer layer is used for conceptual modelling, while the inner layer is optional and used for formal reasoning [Dardenne et al. 93, Darimont and Van Lamsweerde 96]. The KAOS language uses temporal logic [Galton 87, Koymans 92, Manna and Pnueli 92] to describe the inner formal assertions associated with the operations, goals, and invariants. The logical connectives of the temporal logic used within the KAOS language is given as shown in table 3.1:

Table 3.1,Temporal logic notations used within KAOS [Letier 01]

Symbol	Meaning
●	In the previous state (unary operator)
♦	Sometime in the past (unary operator)
■	Always in the past (unary operator)
U	Always in the future until (binary operator)
○	In the next state (unary operator)
◊	Sometime in the future (unary operator)
□	Always in the future (unary operator)
W	Always in the future unless (binary operator)
$\Diamond_{\leq d}$	Sometime in the future within d units (unary operator)
$\Box_{\leq d}$	Always in the future within d units (unary operator)
⇒	Implies (binary operator)
∨	Logical or (binary operator)
∧	Logical and (binary operator)
¬	Logical negation (unary operator)

For example, the temporal logical expression $\Box(C \Rightarrow \Diamond_{\leq d} T)$, where *C* and *T* are two Boolean expressions and d is a time interval, has the meaning: it will always be the case in the future that the truth of the expression C implies the truth of *T* within *d* time units as an upper

bound; the unary □, in the outermost scope, is usually implied and removed from the temporal expressions.

3.2.4 KAOS Goal types

As mentioned before, each goal within the goal-model has both a formal and an informal description. The formal description for each goal is considered as a conditional assignment or achievement in reference to the future. Each goal has a condition required to fire its action, and its action can be to achieve, avoid, maintain, or cease some condition/state. Goals in KAOS are classified into the following four categories:

3.2.4.1 Achieve

The achieve category is meant to describe the accomplishment of a particular task within a time period. It has the following structure: $C \Rightarrow \Diamond_{\leq d} T$ where C is the condition and T is the property to be achieved.

3.2.4.2 Cease

The cease category is meant to describe stopping something within a certain time period. It has the following structure: $C \Rightarrow \Diamond_{\leq d} \neg T$ where C is the condition and T is the property to be ceased.

3.2.4.3 Maintain

The maintain category is meant to describe the maintenance of a certain property of the system for all future time. It has the following structures: $C \Rightarrow T\ W\ N$ or, alternatively, $C \Rightarrow T$, where C is the condition, T is the property to be maintained and N is the stopping condition after whose achievement T is not required any more. The maintain goals can be used to specify safety properties that must be maintained over a period of time, like increasing the lifetime of devices by imposing constraints on their use.

3.2.4.4 Avoid

The avoid category is intended for avoiding certain situations for the system at any future time. It has the following structures: $C \Rightarrow \neg T\ W\ N$ or alternatively $C \Rightarrow \neg T$ where C is the condition, T is the situation to be avoided and N is the stopping condition whose achievement implies that T is not required to be avoided any more. The avoid goals can be used to describe the avoidance of hazard situations.

3.2.5 Goal-oriented Requirements Elaboration

The KAOS method prescribes some steps to generate finer specifications from the stated high-level goals. Figure 3.3 shows these steps. The first step (Goal elaboration) needs recursive processing, for it builds up the goal structure. The steps are ordered by data dependency; they may run in parallel, but with backtracking at every step [van Lamsweerde Y2K].

Step 1: Goal elaboration

This is the very first step of the methodology; it acquires the goals known by the system engineer. From these goals, it identifies the different objects concerned with the system objectives.

Step 2: Object Capturing

Using the goals stated in the previous step in addition to the knowledge given about the system, the objects involved in goal formulation are identified.

Step 3: Operationalization

The Operationalization step derives strengthened (adding more conditions to the logical expression) pre-, post, and trigger conditions on operations, and strengthened invariants on objects.

Step 4: Responsibility assignment

This step accomplishes the following:

- Identify alternative responsibilities for terminal goals.
- Make decisions among refinement, operationalisation, and responsibility alternatives, so as to reinforce non-functional goals [Yu and Mylopoulos 98] (for example, goals related to reliability, performance, cost reduction, life time maximisation).
- Assign operations to agents that can commit to guarantee the terminal goals in the alternative selected. The boundary between the system and its environment is obtained as a result of this process, and the various terminal goals become requirements or assumptions depending on the assignment made.

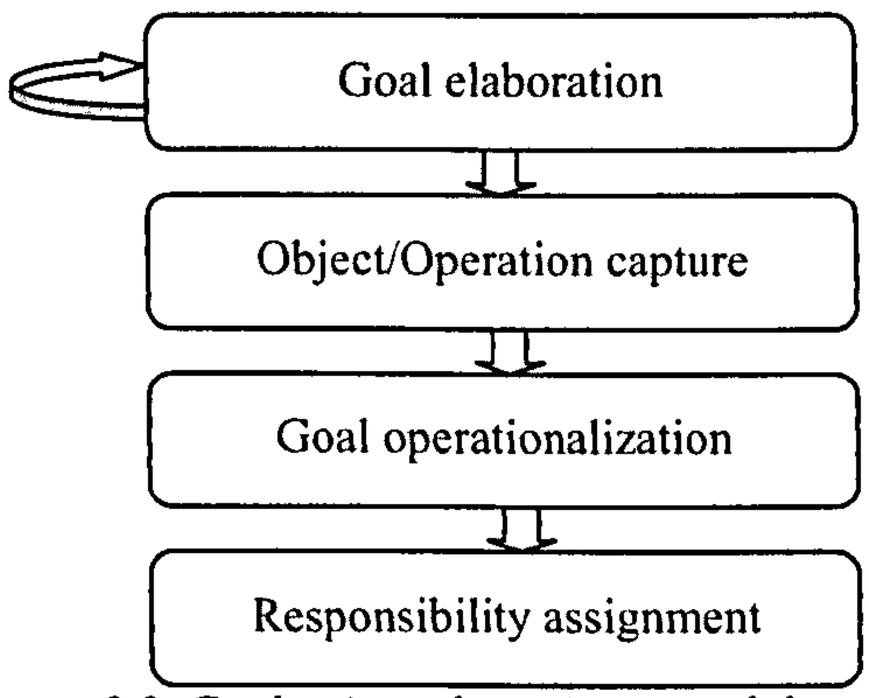


Figure 3.3, Goal-oriented requirement elaboration

The steps shown in figure 3.3 can be viewed differently if the objects and the relationships between them can be identified before formulating the main goal; in this case the goal elaboration can be considered as the second step. Then, the third and fourth steps can be regarded as shown in figure 3.3.

3.3 Goal Conflicts [Van Lamsweerde et al. 98b]

Consistency of the requirements is a crucial aspect [Easterbrook 94, Easterbrook and Nuseibeh 95, Heitmeyer et al. 96, Hunter and Nuseibeh 98, van Lamsweerde et al. 98b]; it should be detected and resolved within the statement of user needs or the refined requirements. Ignoring the inconsistency within the requirements may result in poor implementation.

In large-scale systems, it is more likely that, different types of inconsistency might appear within the system. This happens due to different reasons, such as process-level deviation, instance-level deviation, terminology clash, designation clash, structure clash, or conflict [van Lamsweerde et al. 98b]. The process and instance level deviations are concerned with inconsistencies between the process-level rules or the instance-level rules, respectively, while terminology and designation deals with defining real-world concepts from different views. The structure clash occurs when a

single real-world concept is referred to with more than one structure. Finally, conflict arises as a result of localising the formal description for each goal and neglecting situations when two or more goals can be fired simultaneously. When the number of goals are relatively high, It is more probable that two or more goals conflict. For example, the systems engineer may define a specific goal G_n that changes a particular variable V's value to v_1 under certain condition C_n, while within the same goal-model, there exists another goal G_m that changes the same variable V's value, but to a different value v_2 under a different condition C_m. The conflict problem appears when the two conditions C_n and C_m are satisfied together; therefore, the variable is supposed to be changed to two different values v_1 and v_2 at the same time.

3.3.1 Formal definition of goal conflict

A conflict between assertions A_1, A_2, .., A_n occurs within a domain Dom if and only if the following conditions hold:

$\{Dom, B, \wedge_{1\le i\le n} A_i\} \models$ false (logical inconsistency).....**3.1**

For all i $\{Dom, B, \wedge_{i\neq j} A_j\} \not\models$ false (minimality).....**3.2**

where *B* is the boundary condition within the domain. The condition *B* can vary from time to time depending on the domain variables.

3.3.2 Divergent goals

Goals G1, G2, .. , Gn are said to be divergent goals if and only if there exists a boundary condition that makes them logically inconsistent with each other in the domain considered [van Lamsweerde et al. 98b].

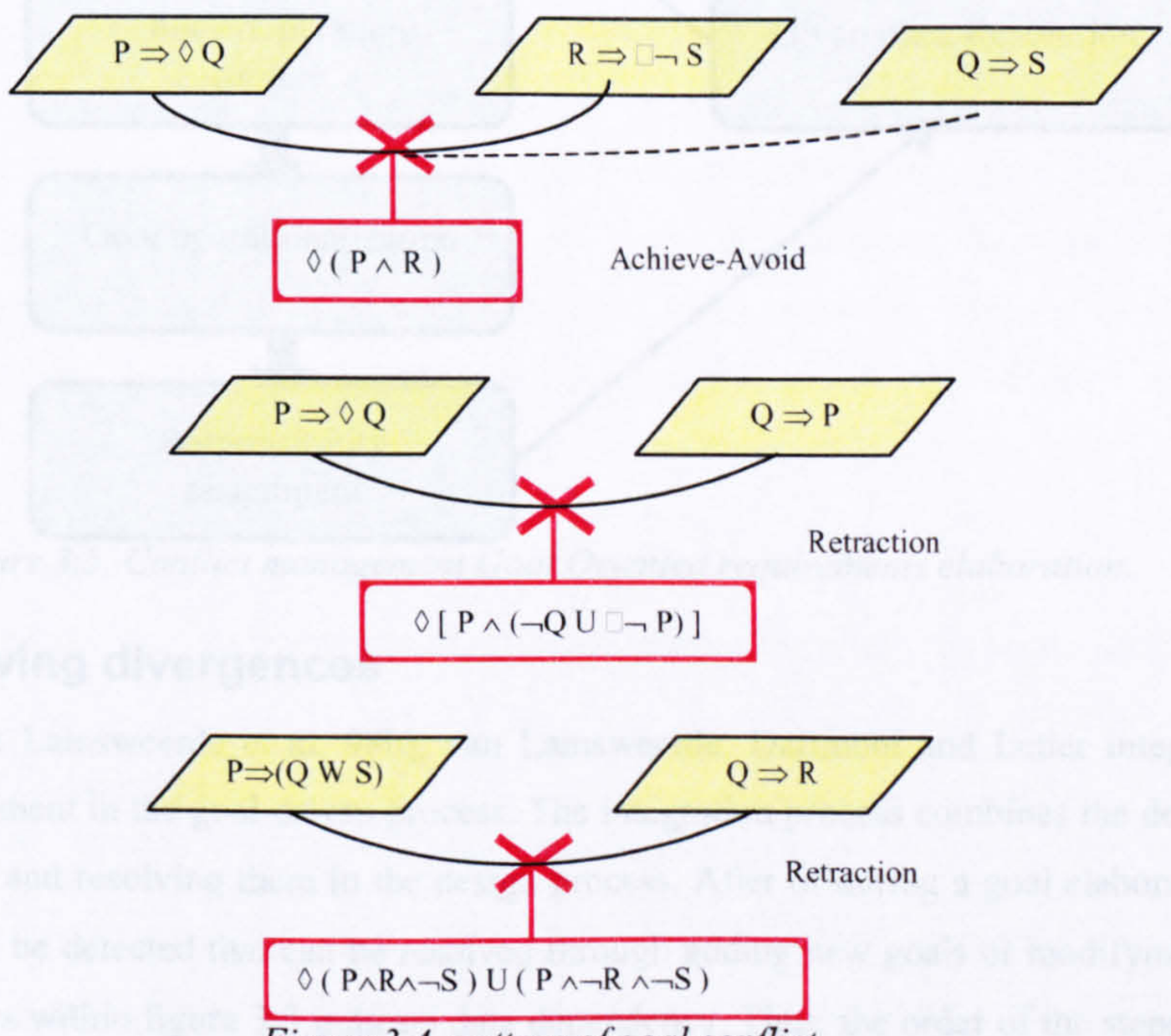


Figure 3.4, pattern of goal conflict

3.3.3 Detecting Divergences

van Lamsweerde, Darimont and Letier in [van Lamsweerde et al. 98b] proposed two techniques for detecting divergences: The former derives boundary conditions by backward chaining; the latter relies on the use of divergence patterns.

3.3.3.1 Regressing Negated Assertions

This technique is based on the condition {Dom, B, $\wedge_{i\neq j}$ A_j} |-- $\neg A_i$. Given some goal assertion A_i., it consists of calculating a pre-condition for deriving the negation of the $\neg A_i$ backwards from the other assertions conjoined with the domain theory. Every precondition obtained yields a boundary condition. The weakest pre-conditions may be considered as they cover most of the general combination of circumstances causing conflict.

3.3.3.2 Divergence Patterns

The other technique of identifying divergence is to use patterns of divergence that can be identified within similar systems; for example, a common pattern of divergence is found between achieve and avoid goal types as shown in figure 3.4; in the upper part, the source of divergence is the goal $\Diamond(P \wedge R)$. There are other patterns of divergence as shown in figure 3.4. The use of patterns is very helpful to alert the systems engineer to sources of inconsistencies that can be found in the requirements but have not been detected yet.

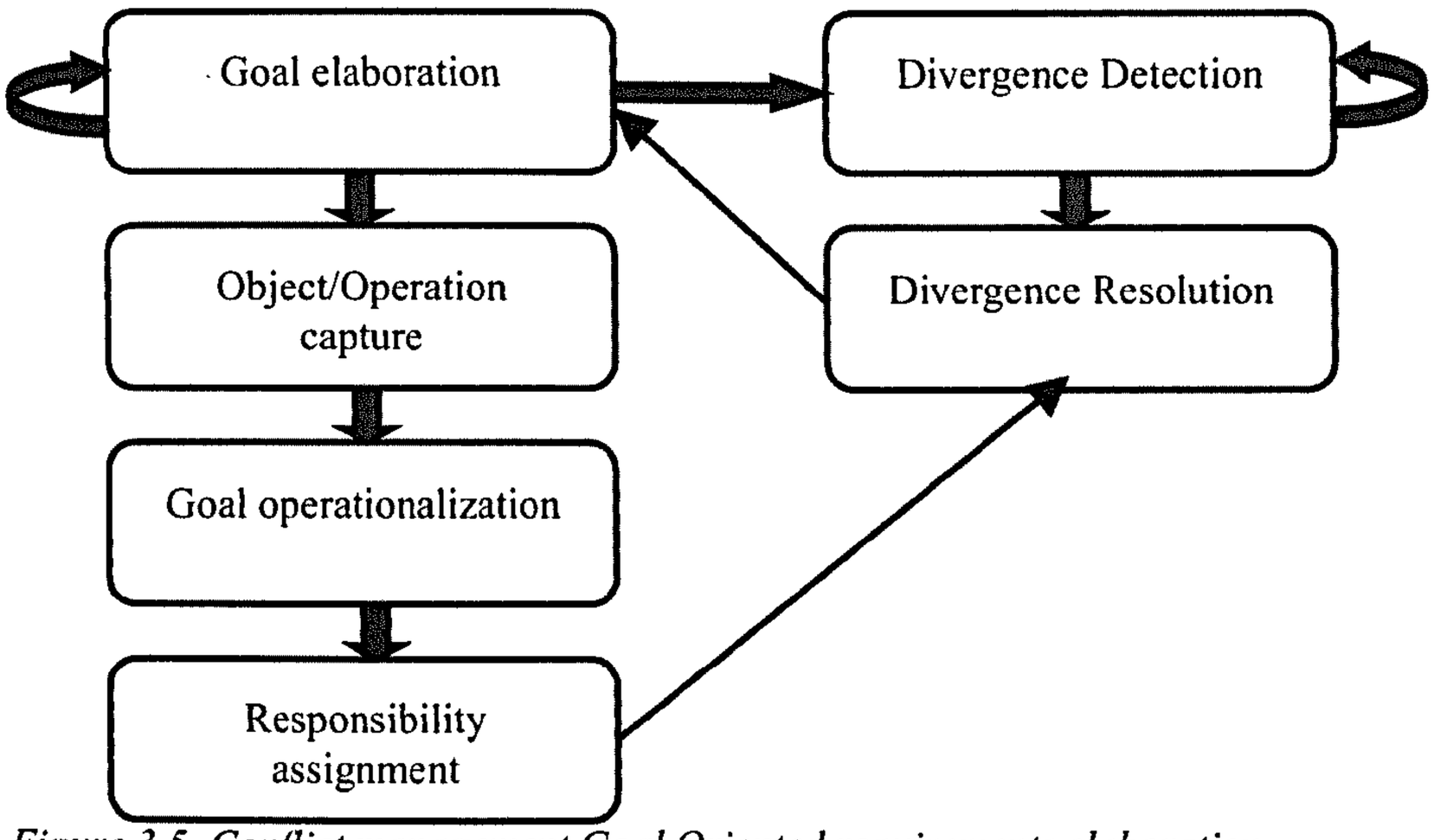


Figure 3.5, Conflict management Goal Oriented requirements elaboration.

3.3.4 Resolving divergences

In [van Lamsweerde et al. 98b], van Lamsweerde, Darimont and Letier integrated the conflict management in the goal-driven process. The integration process combines the detection of the divergences and resolving them in the design process. After or during a goal elaboration step, divergences can be detected that can be resolved through adding new goals or modifying existing ones. The arrows within figure 3.5 indicate data dependency. Thus, the order of the steps can vary provided that there will be checks before continuing with any dependant step; for example, the responsibility assignment step depends on the goal operationalization step. The Divergence

Resolution step can be accomplished through different means: either by changing the precondition of some of the goals to make them more specific, thus preventing conflict with other goals, or through adding new goals to the goal model.

3.3.5 Example of Goal Conflict

Consider the goal model in figure 3.6. The goal model represents a tank system that is used to maintain a liquid level steady between two values Level1 and Level2. The goals G3 and G2 are said to be divergent if the two conditions (liquid level is less than Level1 and liquid level is greater than Level2) together do not imply false. If there is a real conflict between the two goals, then a modification in the goal model is required. Such a modification may loosen the condition of one of the divergent goals.

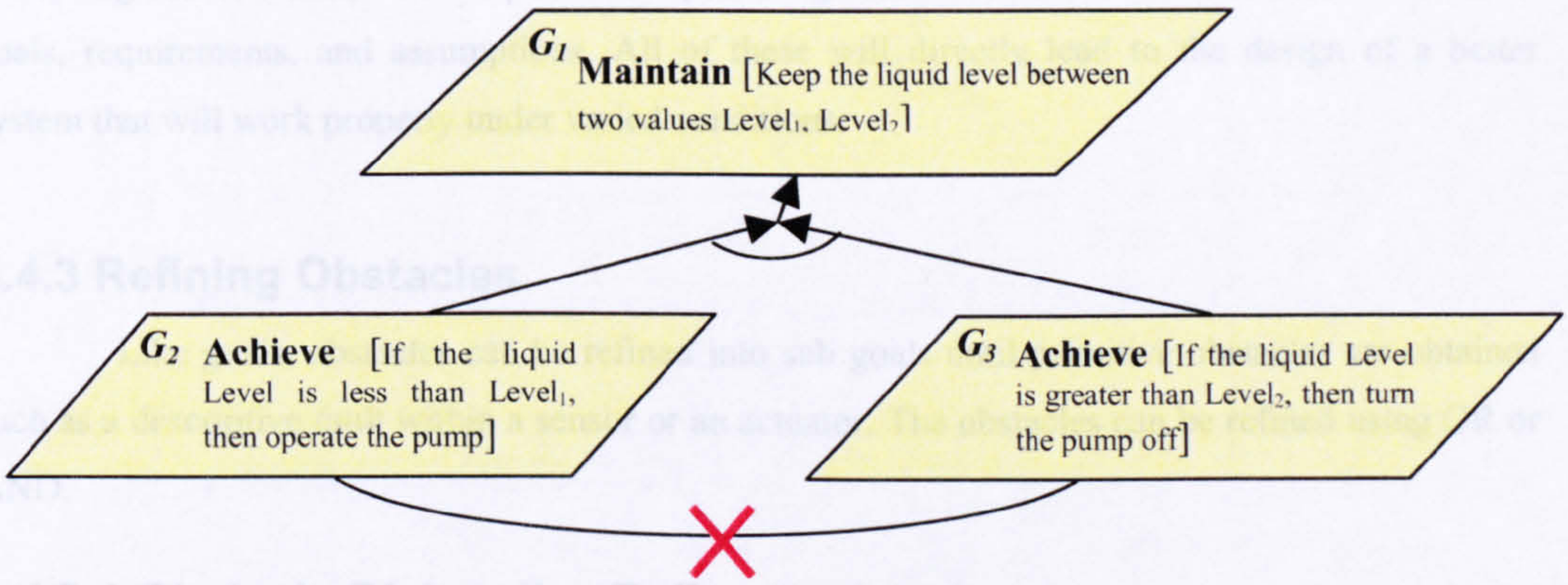


Figure 3.6, an example of probable divergent goals

3.4 Obstacles to Goals [Van Lamsweerde et al. 95]

It may be that at a high-level of system requirements it seems that the goals are acceptable and no obstructions are expected. However, the lower the level of the goal refinement (the less abstract it is), the more obstructions can be discovered. The system goals can be defined as a set of desired behaviours, while obstacles to goals are a set of undesirable agent behaviours [van Lamsweerde et al. 95, Potts 95]. The obstacle types differ depending on the types of goals they obstruct; for example, hazard obstacles obstruct safety goals, misinformation obstacles obstruct information goals, starvation obstacles obstruct satisfaction goals, and threat obstacles obstruct security goals.

3.4.1 Formal Definition of Goal Obstruction

For obstacle O to obstruct Goal G within Domain Dom, the following conditions must hold:

$\{O, Dom\} \models \neg G$ (obstruction)..... **3.3**
$\{O, Dom\} \not\models$ false (consistency) **3.4**
$\exists S$ [scenario(S) $\wedge$ S $\models$ O] (feasibility) **3.5**

The first condition (3.3) means that the presence of the obstacle O will stop the achievement of goal G. The second condition (3.4) means the obstacle does not conflict with the domain. Finally, the last condition (3.5) means the obstacle can occur according to at least one scenario.

3.4.2 Identifying Obstacles

The process of identifying obstacles to the goals can be carried out through the following steps:

- Step 1: Given the goal assumption specification, find some assertion that may obstruct it;
- Step 2: Check that the candidate obstacle thereby obtained is consistent with the domain theory available;
- Step 3: Determine the satisfiability of the candidate obstacle by finding some feasible negative scenario.

The second step is a logical proof that the obstacle is consistent with the domain and can occur with some probability in the normal operations of the system.

Analysing the obstacles with respect to a specific system will lead to more realistic achievable goals, requirements, and assumptions. All of these will directly lead to the design of a better system that will work properly under varied conditions.

3.4.3 Refining Obstacles

Like goals, obstacles can be refined into sub goals until primitive obstacles are obtained such as a descriptive fault within a sensor or an actuator. The obstacles can be refined using OR or AND.

3.4.3.1 Obstacle Disjunctive Refinement

Disjunctive refinement for obstacles indicates that each of the sub-obstacles obstructs the goal G. An obstacle O to a goal G can be refined into disjunctive sub-obstacles O1, O2, .. ,Om. if the following conditions are satisfied:

For all i $\{O_i, Dom\} \models O$ ..... **3.6**(entailment)
For all i $\{O_i, Dom\} \not\models false$ ..... **3.7** (consistency)
$\{\neg O_1 \wedge \ldots \wedge \neg O_n, Dom\} \models \neg O$ ..... **3.8**(completeness)
For all i, j $\{O_i, O_j, Dom\} \models false$ ..... **3.9** (disjointness)

The first condition (3.6) means the obstacle *O* can be derived from each of its sub-obstacles. The second condition (3.7) means each of the sub-obstacles is consistent with the domain. The third condition (3.8) means that the obstacle *O* does not have any other sub-obstacles; and, by negating all of its sub-obstacles, the negation of the obstacle *O* will be reached. Finally, the fourth condition (3.9) indicates that the domain will not be consistent with more than one sub-obstacle.

To stop obstacle *O* from obstructing the goal *G*, each sub-obstacle of *O* must be eliminated as indicated by the second condition.

3.4.3.2 Obstacle Conjunctive Refinement

Conjunctive refinement of an obstacle means the obstacle is going to obstruct the goal G only if all of its sub-obstacles are present. An obstacle *O* to a goal *G* can be refined into conjunctive sub-obstacles $O_1, O_2, .. ,O_m$ if the following conditions hold:

$\{O_1 \wedge \ldots \wedge O_n, Dom\} \models O$ ..... **3.10** (entailment)

$\{O_1 \wedge \ldots . \wedge O_n, Dom\} \not\models false$ ..... **3.11** (consistency)
For all i $\{\wedge_{j \neq i} O_j, Dom\} \not\models O$ ..... **3.12** (minimality)

The first condition (3.10) means the obstacle O will be reached only if all of its sub-obstacles are reached. The second condition (3.11) means that the conjunction of all the sub-obstacles does not produce any inconsistency with the domain. Finally, the last condition (3.12) means that the obstacle O cannot be reached without any of its sub-obstacle. Unlike the disjunctive refinement, to stop obstacle O from obstructing the goal G, any of the sub-obstacles of the obstacle O must be eliminated, as indicated by the minimality condition (the third condition).

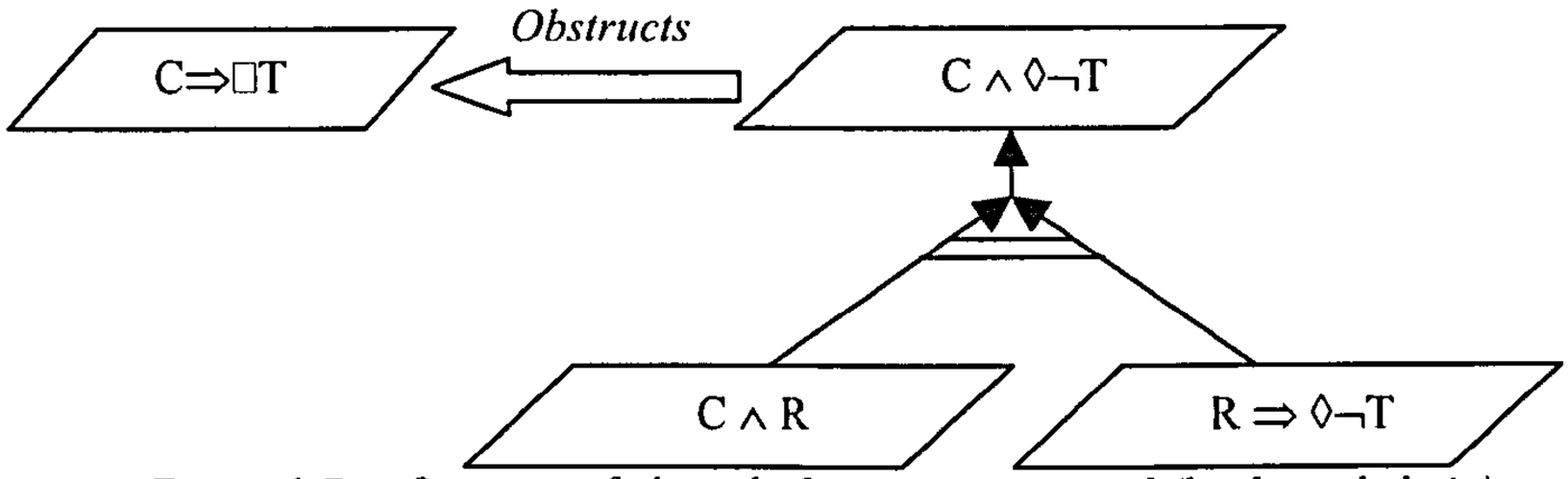


Figure 3.7, refinement of obstacle for a maintain goal (backward chain)

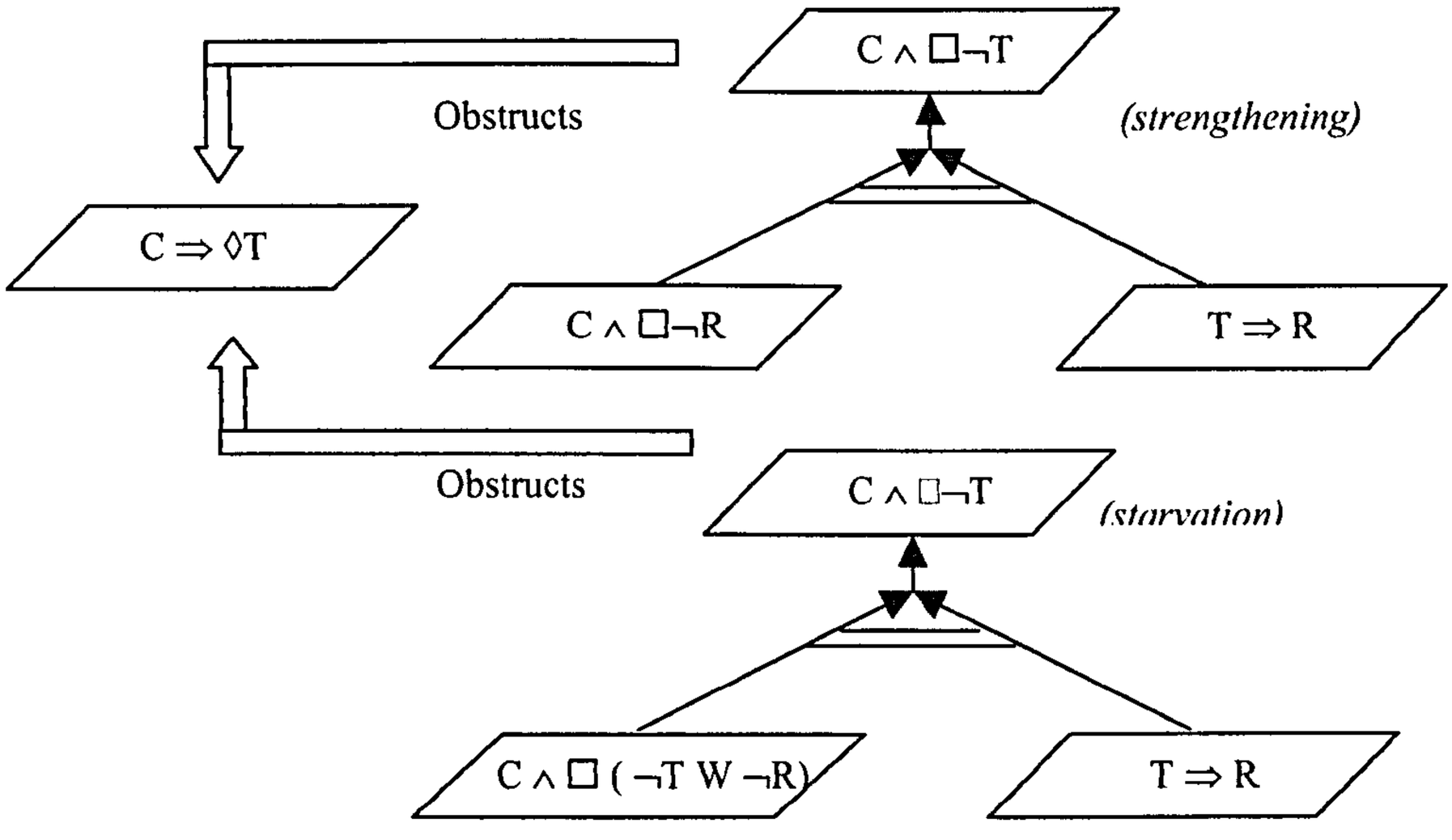


Figure 3.8, refinement for obstacle for achieve goal with form C ⇒ ◊T

3.4.4 Patterns for obstacles according to the goal types

As indicated before in section 3.4.1, obstacles to goals prevent achieving goals through producing inconsistency with the goal; in other words, the internal formal definitions for the goal and its obstacles cannot be true together within the given domain. Thus, in order to discover obstacles for a particular goal, the internal formal description for the goal can be negated to produce a general form describing the obstacles.

For example, for a goal of the form $A \Rightarrow B$, the obstacles to it will have the form $\neg(A \Rightarrow B)$ that can be simplified to $(A \wedge \neg B)$, where $(A \Rightarrow B)$ is equivalent to $(\neg A \vee B)$. van Lamsweerde and Letier in

[Vla98a] used this idea to define the general form for obstacles for the different types of goals as follows:

3.4.4.1 Obstacles to maintain goals

The maintain goals, as already mentioned in section 3.2.3, have the temporal logic form $C \Rightarrow \Box T$, where $\Box$ indicates always in the future. Thus, the obstacles' forms for such goals can be derived by negating the goal's formal description. Thus, the obstacles to maintain goals will have the form $C \wedge \Diamond \neg T$,meaning the condition C will be present while the achieved condition T will not be present sometime in the future. The obstacle of such a form $(C \wedge \Diamond \neg T)$ can be refined as the conjunction of $(C \wedge \Diamond R)$ and $(R \Rightarrow \Diamond \neg T)$. This means the obstacle can be refined to two obstacles with the following semantics: the first obstacle with the condition C will be true with another intermediate condition R, and the second obstacle means this intermediate condition R will stop T from being maintained some time in the future, as shown in figure 3.7. Such a refinement is called a backward chain because we seek a path between R and T starting from T, which is considered as the final goal to be achieved.

3.4.4.2 Obstacles to avoid goals

As for obstacles for maintain goals, for avoid goals of the form $C \Rightarrow \Box \neg T$, the obstacle will have the general form $C \wedge \Diamond T$ that can be refined to the conjunction of $(C \wedge \Diamond R)$ and $(R \Rightarrow \Diamond T)$.

3.4.4.3 Obstacles to achieve goals

The achieve goals, as mentioned in section 3.2.3, have the form of $C \Rightarrow \Diamond T$ where $\Diamond$ means sometime in the future. Thus, the obstacles for such goals will be of the form $C \wedge \Box \neg T$, meaning the condition C will be present while the achieved condition T will not be present at any point in the future. The obstacle of such form $(C \wedge \Box \neg T)$ can be refined in two ways as shown in figure 3.8: first as a conjunction between $(C \wedge \Box \neg R)$ and $(T \Rightarrow R)$, meaning the condition C will be true with an intermediate condition $\neg R$ (where the achieving of T implies the achieving of such an intermediate condition R) that is always false (strengthening) or, second, as a conjunction of $(C \wedge (\neg T \ W \ \neg R))$ and $(T \Rightarrow R)$, meaning the condition C will be true with the condition T false until an intermediate condition R becomes false (where the achieving of T implies the achieving of such an intermediate condition R) (strengthening).

3.4.4.4 Obstacles to cease goals

As for obstacles for achieve goals, for cease goals with the form $C \Rightarrow \Diamond \neg T$, the obstacle will have the following general form $C \wedge \Box T$ that can be refined to the conjunction between $(C \wedge \Box R)$ and $(R \Rightarrow T)$.

3.4.5 Goal model modifications

After the obstacles to some goal have been identified, the goal should be reformulated to ensure its achievement, even in the presence of such obstacles, by weakening the goal and/or adding new goals to prevent/attenuate/ recover from the obstacle.

3.4.5.1 Goal de-idealised

Van Lamsweerde and Letier in [Van Lamsweerde et al. 98a] suggested that the goal could be de-idealised (the pre condition of the goal will be constrained or the goal will be only fired in some of the cases, in which it was meant to be fired)through weakening it. The de-idealisation will consist of two steps:

- Weaken the goal formulation to obtain a more liberal version that covers the obstacles (make the rule more general by adding a disjunct or removing a conjunct).
- Propagate such a change through the goal model (add the same disjunct or remove the same conjunct wherever the old predicate is).

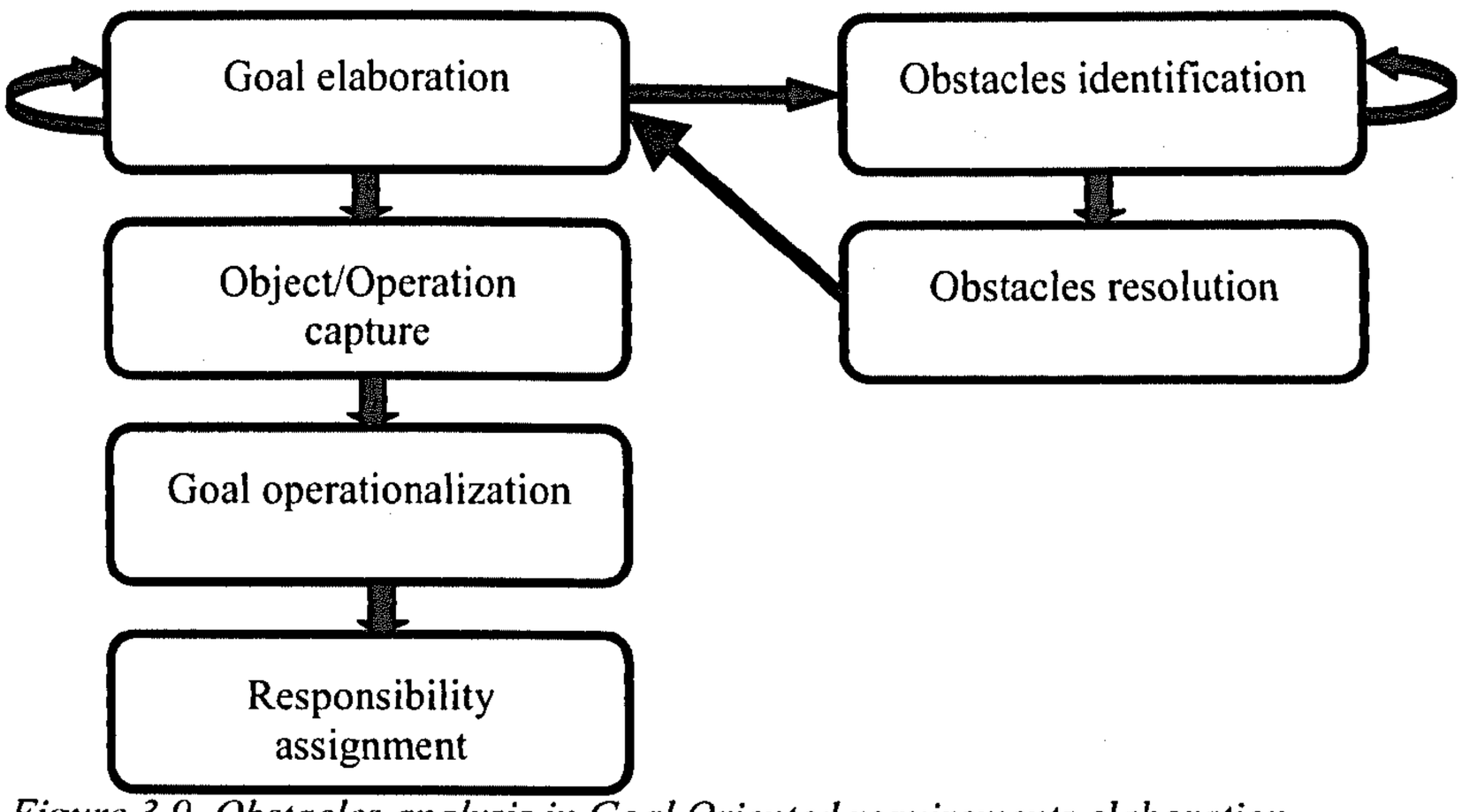


Figure 3.9. Obstacles analysis in Goal Oriented requirements elaboration.

3.4.5.2 Introducing new goals

In addition to changing the obstructed goal, a new goal may need to be added to the goal model. van Lamsweerde and Letier in [van Lamsweerde et al. 98a] suggested three different strategies for introducing new goals to resolve obstacles:

I) Obstacle Prevention

A new goal of type avoid is introduced. The goal has the form $C \Rightarrow \Box \neg O$ where O is the obstacle or a pre-condition leading to it ($C \Rightarrow \Box \neg R$ *and* $R \Rightarrow O$). The presence of such a goal in conjunction with the obstructed goal G will guarantee that the obstacle O will not appear.

II) Goal Restoration

It is often the case that obstacles that result from unexpected behaviours of an agent in the environment cannot be avoided; thus, a first possibility is to restore the obstructed goal from states

in which the obstacle occurs. Or, in other words, to recover the error (obstacle) means adding a goal of the form $O \Rightarrow \lozenge G$ where O is the obstacle and G is the obstructed Goal.

III) Obstacle mitigation

In case of unavoidable obstacles, the idea is to attenuate the obstacle's consequences by introducing new goals. Such goals should minimise the effect of the obstacles. For example, if the Obstacle has a set of consequences *C1, C2, .. ,Cn,* new goals can be introduced to eliminate some or all of such consequences in the future.

During elaboration of the goal tree by elicitation and by refinement, obstacles are generated from goal specifications. Such obstacles may be recursively refined. Resolving the generated obstacles usually results in updating the goal-model through adding new goals and/or modifying the existing ones. The resolution of an obstacle may be subdivided into two steps [Easterbrook 94]: the generated alternative resolutions, and the selection of one among the alternatives considered. Figure 3.9 outlines the modification of goal-oriented requirements elaboration by adding obstacle analysis. The generated obstacles are resolved, which results in new goals and/or transformed versions of existing ones. The new goal specifications obtained after negating the obstacle , may in turn trigger a new iteration of goal elaboration and obstacle analysis. A number of methodological questions may arise about how many obstacles are to be generated and for which kinds of goals: high or low level ones.

3.4.6 An Example of Goal Obstruction

For example, the goal model of figure 3.3 has a main goal of type maintain. Four main obstacles for goal G1 can be observed as follows:

- Failure to increase the liquid level when required (the pump cannot be turned on)
- Failure to decrease the liquid level when required (the pump cannot be turned off)
- Failure in detecting the case of exceeding the upper bound (the higher level sensor malfunctions)
- Failure in detecting the case of dropping under the minimum level (the lower level sensor malfunctions).

Each of these obstacles obstructs goal G1; thus, the obstacle for the main goal can be refined disjunctively into these four obstacles. Figure 3.10 shows the refinement of the obstacle O1 to the main goal. However, having such an obstacle analysis at the level of Goals G2 and G3 instead will prune the number obstructing each goal. Goal G2 is obstructed by two obstacles: either failure to detect dropping below level Level1 (lower level sensor malfunctions) or failure to turn the pump on when required (the pump or the driving motor malfunctions). Similarly, goal G3 is obstructed by a compound obstacle O2 that can be refined to two obstacles O5 and O6, as shown in figure 3.11. In [Van Lamsweerde and Letier 98b], Van Lamsweerde and Letier recommended that the obstacles should be identified from the terminal goals that are assigned to individual agents. Having the obstacle analysis at the terminal goal level saves the effort required to refine the compound obstacles for high-level goals.

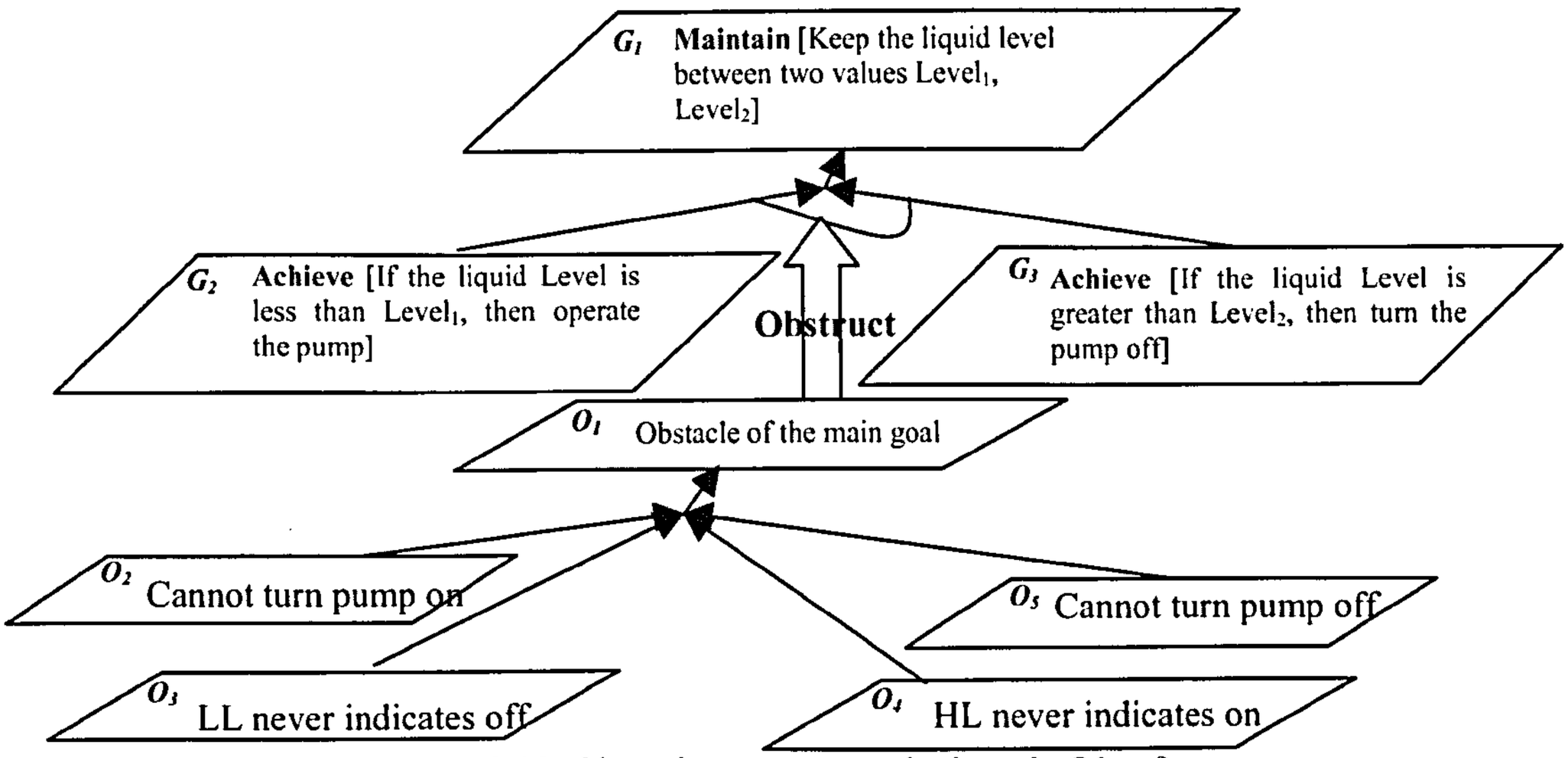


Figure 3.10, Obstacles to main goal, obstacle O1 refinement.

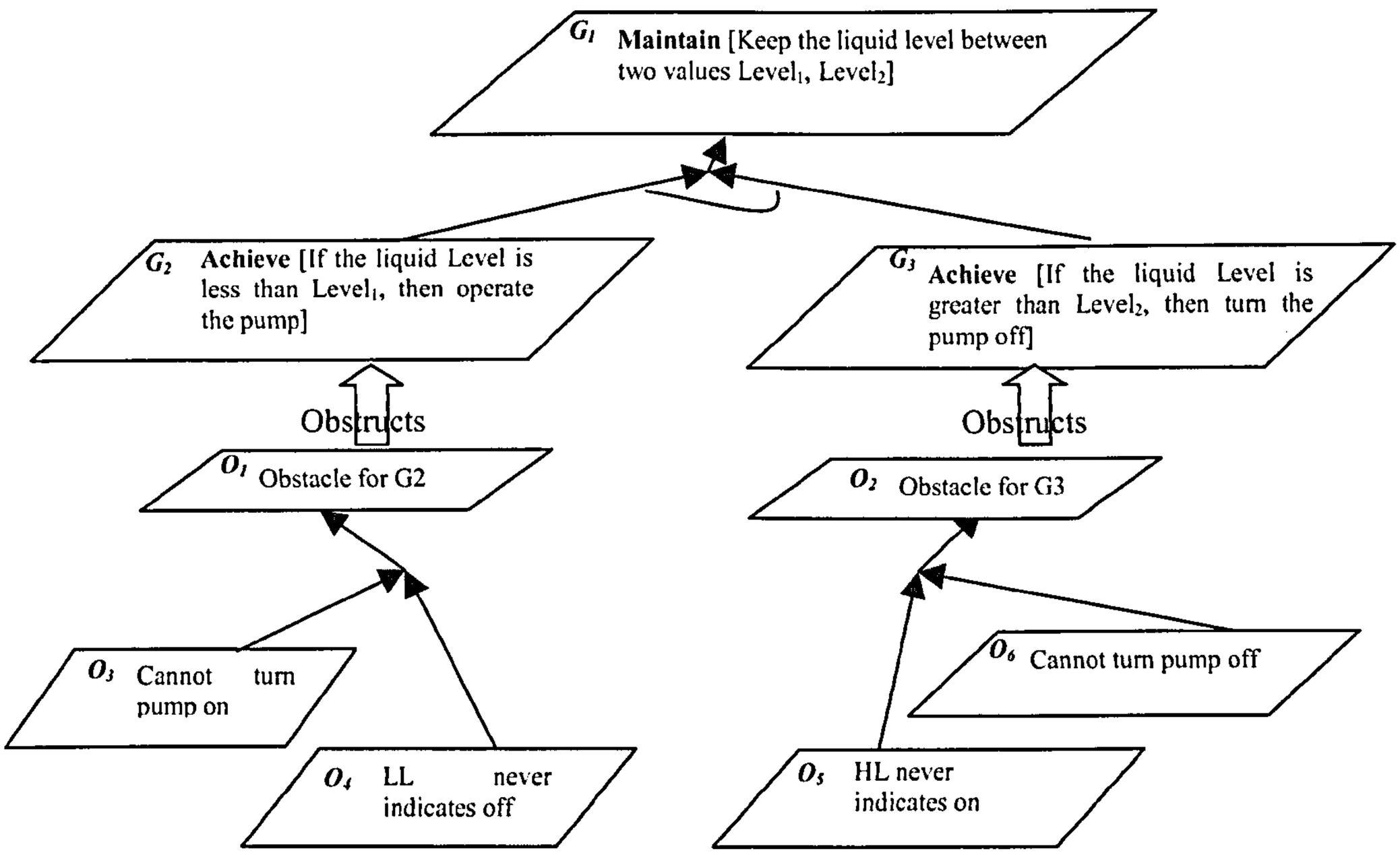


Figure 3.11, Obstacles to the sub-goals.

3.5 Generating Formal Specifications

As previously mentioned, the main objective of the KAOS requirements analysis method is structuring and checking the software applications requirements. Furthermore, the research was extended in order to transform the requirements into useful formal specifications. The specification stage implicitly starts by providing formal descriptions of the functional terminal goals within the goal-model. Letier and van Lamsweerde, in [Letier and Van Lamsweerde 02b], provided techniques for building operational models from goal-oriented system requirements.

The formal descriptions of the terminal goals, in the form of temporal-logic formulae, were used to identify patterns of operations. Each operation has input and output variables and pre- and post-conditions in addition to trigger conditions (like events, that will trigger the execution of

the operations during run-time). The requirements specification stops at the level of operations and generating state specifications, because state charts and operations can be considered as general building blocks of any specification language. As stated in [Letier and van Lamsweerde 02b], deriving the operations in this way has the advantages of abstraction from the formal details, assurance of completeness, guidance in writing operational specifications and goal mining from operational specifications (i.e., using the operationalization patterns bottom-up to elicit undiscovered goals).

Although the operationalization patterns are very helpful, a structure for the sequence of the operations cannot be clearly identified. After constructing, checking and correcting the goal driven requirements, the client would expect more than statecharts or state machines. The client, to gain a sufficient level of confidence, requires a well-defined path towards implementation or the generation of an intermediate specification, that can be used systematically and easily to build the implementation. Hence, further effort is needed to produce a formal specification module in a language like B.

3.6 Other Goal-oriented Requirements Analysis Approaches

In this section, we point to some other efforts focusing on goal driven requirements analysis in the software engineering area. These analysis methods will not be discussed in detail since either they only address the non-functional part of the requirements or only informally address the functional side, whereas the KAOS method can combine the functional and non-functional requirements.

3.6.1 Goal Based Requirements Analysis Method (GBRAM)

In [Antón 96, Antón 97], Antón introduced a goal-based requirement analysis method (GBRAM) for identification and refinement of goals into operations for software based information systems. Following the introduction of the method, she developed a web-based tool (GBRAT) [Antón 95] to support the method.

The GBRAM research presents four heuristics to identify, refine, classify and elaborate goals. The method provides informal semantics for goals and thus does not support formal reasoning.

3.6.2 Goal-Oriented Requirement Language (GRL)

At the university of Toronto, Cyseiro and Liu [GRL] developed a goal-oriented requirement language (GRL) that supports goal and agent orientations and deals mainly with non-functional requirements like security, safety, understandability and cost. They model different concepts like goals, tasks, beliefs, resources and soft-goals; these concepts are related to each other via decomposition, dependency, contribution and correlation.

Through building goal-model like structures, the application can provide answers for "Why", "How", and "How else". They applied the modelling language to the security issues of both smart card systems and developed software applications.

3.6.3 The TROPOS Methodology

The TROPOS framework [Mylopoulos Y2K, Giorgini 02] is an approach based on agent-oriented software engineering that starts at the very early requirements stages. A transformation approach is adopted to enable the refinement steps to be performed inside one phase or between phases using some transformation operators. The phases covered by the TROPOS methodology include analysing and modelling the environment in terms of relevant actors, introducing system actors and analysing dependency between them, defining the global architecture in terms of subsystems and, finally, defining detailed design, agents, components and other relevant information.

One important aspect of the TROPOS methodology is that it combines refining the system goals and evaluating the different alternative solutions. The evaluation is based on how much high-level goals are satisfied by the use of propagating algorithms. The methodology was used to choose an alternative solution for a car design company. However, the evaluation process considers the contribution between the different goals and does not address directly the case when refining a hard (not soft) main goal.

3.7 Conclusions

The requirements analysis stage is of great importance for agreement between the client and the service provider. Detecting requirement inconsistency and incompleteness reduces considerable time and cost that would be required in later stages to correct requirement errors. The software requirement specifications should possess some properties such as completeness, consistency, and modifiability, in order achieve success in developing and maintaining the software applications. Goal driven requirements analyses combine precision and rationality. The goal-driven methods utilise requirements models, which satisfy most of the well-definedness attributes of software requirements specifications. Thus, the systems engineer can use these methods to identify the application related objects and then develop the application requirements step by step.

The goal driven requirements analysis method of KAOS provides consistency and obstacle analyses for the developed goal-models. This enables the systems engineer to discover the hidden bugs within the unchecked requirements, as well as to take into account unanalysed situations that might arise during runtime. The validated requirements can be formulated as formal specifications for the intended application. There are other goal-oriented requirements tools; the ease and reasonability of the hierarchical nature of the goal-model provides an easy and normal approach for specifying application requirements in general.

Formal Methods 4

In this chapter, we introduce formal methods and the role they play in the life cycle of software applications. The main concepts of the algebraic specification, Z, VDM and B methods are described, with more stress on B and its associated toolkit; an example of a gas burner system illustrates the process of developing reactive system specifications in B.

4.1 Introduction

Formal methods provide a setting for specifying software applications and ensuring separation of concerns between the implementation needs of the applications and their abstract specifications. This separation can help in dividing the effort required to develop a software application, as well as providing traceability to more easily modify and maintain the application. Hence, formal methods enable the software system designer to reason about the decisions that are taken to develop the application. This motivates why most formal methods are described by such a phrase and why they are based on mathematical and logical concepts. This sophisticated formal nature enables formal proofs about the application specifications, which encourages testing and checking of the specifications independently from the detailed implementations. And this in turn, enables multiple simple tests and checks at different stages rather than a single complicated check and test at the implementation level.

Thus, the GOPCSD method can serve as a front end to the formal method, translating the abstract requirements decisions agreed by the systems engineer to formal specifications

readable by the software engineer, who can more easily use the formal methods tools to further refine and develop the control application.

Within the formal methods, the application requirements are formulated as specifications, which combine features of programming languages (like being formal in defining variable assignments or control structure statements such as *if* and *while*) and features of the requirements (flexibility and specifying what to do but not how to do it). Such a mixed nature allows formal methods to formalise the requirements and then to construct the outlines of the implementation programs. In order to achieve this, specifications use mathematical notations to describe the properties that a system must have, without excessively constraining the way in which these properties are achieved. This enables the formal specifications to describe what the system must do without saying how it is to be done to separate the formulated requirements from the implementation details of the applications [Spivey 92].

Some of the formal methods, such as Z, VDM, and B, are considered to be 'model-oriented'. These methods are concerned with modelling behaviour of physical or logical systems using some basic, predefined data types and constructions. In contrast, OBJ is considered a property-oriented formal method, mainly defining logical properties as the main concern, without using a predefined set of data types and constructions.

Formal methods have been employed extensively in developing safety-critical software applications, either entirely or partially. These applications require the implementation to conform to an initial design or specification so as to preserve safety properties. As noted in [Spivey 92], *it is important to notice that formal methods do not develop the software applications but provide a stage for the user, in which he/she can determine what are the consequences of his/her decisions before realising these decisions at the implementation level; and be able to correct and complete the specifications before it will be difficult to detect and modify these problems at the implementation stage.*

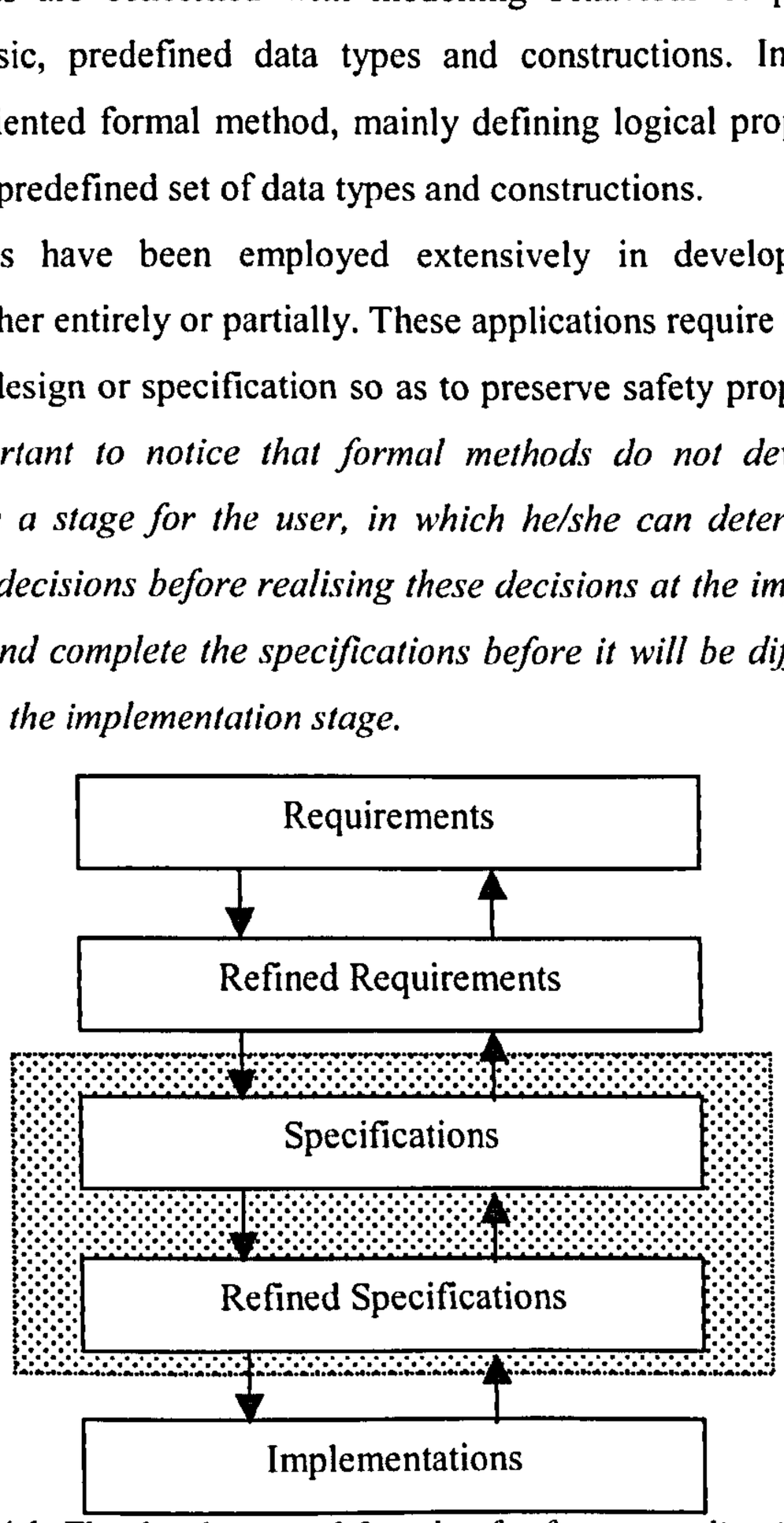


Figure 4.1, The development lifecycle of software applications

Figure 4.1 shows the development lifecycle of software applications. The lifecycle starts by collecting the informal requirements/user needs from the client. Following this step, the software engineer usually refines the requirements in cooperation with the client. The

informal requirements will be then translated to formal specifications in mathematical and/or logic notations. Because these specifications constitute only the outlines of the implementation programs, further refinement may be needed to construct the final implementations. Within the figure, the upward arrows are used to indicate the possible modification and/or correction paths that may be required after performing tests, validations or verifications. These validations and verifications can be manual, automated, or interactive with the user to examine arbitrary scenarios. After the software application passes the implementation and integration tests, it should be maintained for any further extensions or against any changes in the environments where it is supposed to operate.

Formal methods usually provide tools to translate the specifications into high-level languages such as C and JAVA or, in some other cases, machine code. In addition to the translation, the formal methods often provide animation and generation of proof obligations about the application design. These proofs provide an easier, earlier and less expensive chance to correct errors made in gathering, refining or formalising the requirements.

4.2 Algebraic Specifications

One of the earliest algebraic specification languages developed was Clear [Burstall and Goguen 80], which introduced the notations of parameterised specifications; following that, other early languages include OBJ0.

OBJ was designed in 1976 by Goguen [Goguen Y2K], using error algebras to extend algebraic data type theory with error handling and partial functions.

As explained in [Goguen Y2K], the most important OBJ unit, which is considered as the building block, is the object. The object can be used to specify physical and logical components, like a stack of numbers, a liquid tank or controller. The object is defined using the *obj* keyword as follows:

```
obj <object-name> is
...
endo
```

where object-name denotes the object to be defined; it can be a single identifier like *stack* or *liquid_tank* or it can have parameters as in *Robot(position)*

Within the object body, the user defines variables using the *var* keyword and operators, using the *op* keyword, including prefix, postfix, or infix operators; or as functions (methods), as for example, the following *length* operator:

```
op length _ : List -> Int .
```

This specifies an operation to count the number of elements in an integer list of *List* type.

A strong aspect of OBJ is the ability to express equations rather than the common assignment statements; equations are usually used as invariants, or to be reformulated into assignment statements. For this purpose, OBJ uses the *eq* keyword to specify the required equations, as shown in the following example:

```
eq length I = 1 .
eq length ( I L ) = 1 + length L .
```

These two equations state that the operator (method) *length* returns 1 when applied to a list consisting of a single integer value; otherwise, when applied to a list consisting of more than one integer, it returns one plus the length of the rest of the list. The different features of the OBJ specifications can be mapped to programming constructs, such as functions, function-invocations, and . For example, lists of integers can be specified as an object in OBJ as follows:

```
obj LIST-OF-INT is
      sort List .
      protecting INT .
      Subsort INT < List .
      op _ _ : Int List -> List .
      op length _ : List -> Int .
      var I : Int .
      var L : List .
      eq length I = 1 .
      eq length ( I L ) = 1 + length L .
endo
```

where length is a method that returns the number of integer elements found in the list, while the two equations together define the length of the list recursively; the first empty operator?????? is used to define that an integer put in front of a list constitutes another list.

Finally, OBJ uses the *reduce* keyword to debug the algebraic specifications, in order to provide the user with an early validation stage; for example, to compute the length of an integer list like (1 -2 13), the user can type in:

```
reduce length(1 -2 13)
```

And, as a result the following rules will be applied

```
length (1 -2 13) =>
1 + length (-2 13) =>
1 + (1 + length 13) =>
1 + (1 + 1)=>
1 + 2 =>
3
```

This trace of computations should guide the user to debug the specifications and to discover odd cases where the specification may need some modification. Thus, the user can be guided to modify the specifications by guarding the occurrences of such special inputs.

Algebraic specifications have the advantage of being close to the very low-level informal requirements; however, they are not convenient for specifying large-scale applications, where formal specification may be required at an abstract level.

4.3 The Z Language

Abrial developed the Z method in early eighties at the University of Oxford; then in 1990, Spivey gave an initial formal semantics for the method. The Z language is based on set theory and predicate calculus. A specification in Z comprises a set of schemata, where each schema defines the relationships between specific entities. The basic form of a Z schema is as follows:

```
Schema name
Schema Signature
Schema Predicate
```

For example, the following schema with name *Oil*, describes three quantities: *Oil_pressure*, *Max_pressure* and *Min_pressure*. The body of the schema contains two predicates connected by logical conjunction. The body of the oil schema defines a constraint for the *Oil_pressure* value that must be between *Min_pressure* and *Max_pressure*.

```
Oil
Oil pressure: N
Min_pressure: N
Max_Pressure: N
Oil_Pressure > Min_pressure ∧ Oil_Pressure <Max_pressure
```

In the Z language, a schema can reference one or more of the existing schemata. This allows simpler hierarchical description of the described system. Although Z specifications have a mathematical nature based on set theory, the schema of the Z language cannot fulfil the role of component, to represent a physical or logical part of the control system. Moreover, there is a few supporting tools for refinement and verification of Z specifications. This discourages developers to specify the entire software application in Z.

4.4 Vienna Development Method (VDM)

IBM's Vienna research laboratories in Austria developed VDM [Jones 90] for specifying the behaviour of a system, along with techniques for producing designs and programs that conform to this specification. VDM specifications are composed of operations that have state consisting of external entities with which it interacts. VDM permits the use of conditional predicates such as, WHILE and IF.. THEN.. ELSE; It may also have pre- and post-conditions. The basic form of a simple operation in VDM is as follows:

```
NAME
ext  <list of external variables>
pre  <list of pre-conditions>
post <list of post-conditions>
```

An external variable may be read only (rd) or read and write (wr). Read only variables are not controlled by the program, but are only observed, while read and write variables can be declared as shown:

```
ext rd  temperature : N
    rd  pressure    : N
    wr  control     : R
    ...
```

The pre conditions can be simple conditions or compound conditions as follows:

```
pre
temperature > 0 and
temperature < 255
```

Post conditions describe the effects of operations; they can be applied to more than one *wr* variable as follows:

```
post
output = (temperature - offset )
and
error = (temperature - req_temperature)
```

As noted in [Hinchey and Bowen 95], VDM was initially used in compiler definition and in related programming language areas such as database management systems, garbage

collection, and heap storage. However, Z and B are arguably more comprehensive than VDM, resulting from the richness of mathematical types and easier syntactic construction for including smaller specifications in larger ones. Like the Z method, there are some difficulties within VDM in structuring the applications into components. Although VDM and Z are widely used for sequential applications, both have difficulties with concurrent systems. Notations for expressing concurrency are normally based on temporal logics or process algebras.

4.5 The B Method

J. R. Abrial and the research group at BP Research, MATRA and GEC Alstrom developed the B [Abrial 96] language in the 1980's. B is a model-oriented method based on set and refinement theories. B has been designed to cover most of the development phases of the software lifecycle, from specification to implementation, with special emphasis placed on modularity and data encapsulation.

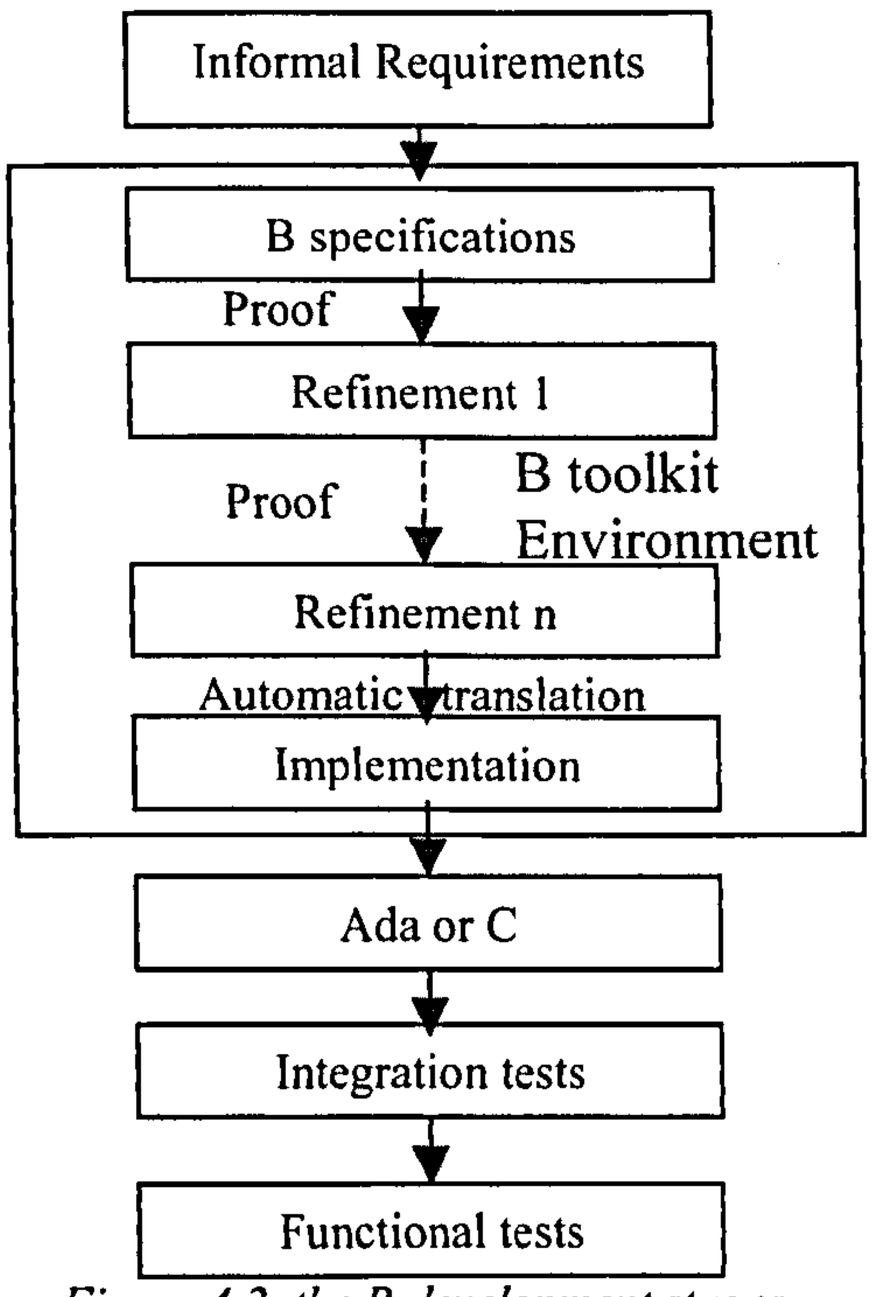


Figure 4.2, the B development stages

The B language has an abstract machine notation AMN; it models systems as abstract state machines built from components, each of which consists of a state, a list of variables, and state-transitions, a list of operations that change the component state through assigning different values to the variables constituting the state. Each component can be specified in a separate module called a machine. The B language allows different forms of interaction between the different machines to build hierarchical systems.

The B method is based on an abstract machine theory and a refinement theory. The development process using B usually takes several steps. The first step produces an initial abstract specification machine(s) for the application. And, each of the following steps refines the outcomes of the previous step. This gradual adding of detail facilitates the construction of

the program step by step and enables the user to have backtracking points to return for debugging, evolving, or maintaining the application's design. In section 4.7.4, we will provide further details of refining B machines. From the above, one can see that the B method can model the system or the application component by component and the relationship between components in a hierarchy. It can also refine these machines gradually (in a single step or more, which can make the refinement process easier to perform and decrease the chances of producing errors/deviations from the desired behaviour). Thus, we have chosen the B method as the main output format to which to translate the corrected requirements from the GOPCSD method.

An abstract machine is a module encapsulating both constant and variable data and operations on that data. The Lifecycle of the software application starts by collecting the informal requirements and then formulating them as B specifications. These specifications may need several refinements to complete the details of the operations in gradually increasing detailed stages; each refinement step needs a proof in order to ensure that it conforms to the previous step. Figure 4.2 illustrates the software application lifecycle using the B toolkit as the integrated development environment for the specifications. The user has to provide the specification in the form of formal requirements (B specification machines); then, within the B toolkit environment, he will have the ability to analyse, refine and implement the specifications and finally translate it into final executable code.

4.5.1 The B Language

The B language has syntax similar to a programming language like PASCAL, and JAVA [Lano and Haughton 96, Wordsworth 96]. In B, the software applications are represented as specifications, refinements and then implementations, in different development phases. Specification is the first phase to represent the application requirements, while the refinements are more detailed versions of the specification machines. Finally, the implementations are produced when every single detail is fully determined, making the implementations ready to be translated into high-level programming languages.

Like high-level programming languages, B has the features of data hiding, data encapsulation and a basic type of sole inheritance using INCLUDES or EXTENDS, to enable data reuse and break down the code into simpler reusable modules. The main unit for specification in B is the machine. Each machine has a name, variables, constraints, invariants, initialisation and operations. Each B machine is considered as an abstract machine that specifies a program by means of an internal state, an initialisation of the state and a number of operations to change the machine state in a manner that obeys the constraints and preserves the invariants. The state specifies the static properties of the program; the state is defined by the collection of variables characterised by an invariant. The invariant is a conjunction of first order predicates of a typed set theory. The syntax of the B machine is shown as follows in figure 4.3.

```
MACHINE Machine name (parameters)
CONSTRAINTS
      Parameters' properties
```

```
SETS
      Local "types"
PROPERTIES
      Definitions of sets and constraints
VARIABLES
      Local variables
INVARIANT
      Properties of variables
INITIALISATION
      Description of how the initial state is created

OPERATIONS
      Operation_name =
      Operation body
      END
   :
END
```

Figure 4.3, the basic structure of a B machine.

The details of each part of the B machine specification are given as follows [Lano and Haughton 96, Abrial 96]:

- **Machine**

This part defines the machine name and any required parameters. For example, a tank can be specified using an integer parameter (to indicate its capacity) to a machine called tank. In addition to the name and parameters, this part can include uses, sees, or includes clauses in order to indicate the relationship between this machine and other machines of the system.

- **Constraints**

This part and the following parts of sets, constants and properties describe the static structure of the machine data, provided that it will remain static and not be changed by the machine operations.

- **Variables**

The variables of the machine are similar to the data of an object in object-oriented languages. Variables appear in this section but the linkage between them and the values assigned to them is represented within the invariant section.

- **Invariants**

Invariants represent the conditions that the machine variables must obey during the lifetime of the machine's operation. For example, they can describe the range of values for the variables and any restriction that must be satisfied by the machine variables at any time. These invariants must be satisfied at the variable initialisation (initial state of the machine) and maintained by each operation of the machine.

- **Initialisation**

This section sets the initial values for each variable; this part is essential for the proofs of properties because to prove any property in B, the proof always starts with the assumed initial values.

- **Operations**

The operations of the machine are considered as the interface with the outside world. They may have parameters and can return data from different types. Each operation will have a pre-

condition to determine the initial state before performing the operation, whereas the post-condition will be described through a single assignment statement.

B offers various data types and structures such as Boolean, integer, Cartesian products, sets, power sets, relations, functions, sequences and trees. This variety provides the user with flexibility when specifying the programs. Both the initialisation and operations are expressed as sets of substitutions that update the state variables of the machine under certain pre-conditions. An abstract machine has to be proved consistent; this implies that the initialisation must satisfy the invariant and the operations must preserve the invariant. This satisfaction and preservation are the proof obligations associated with the abstract machine.

The B language has a very rich set of notations for set theory and non-determinism to allow flexibility in describing what the operations can do as pre- and post- conditions even if the details of this achievement are delayed to later refinement or implementation stages. As mentioned before, in B specifications, the machines are considered as the building blocks; the application is usually built as a hierarchy of machines in a similar way to modular programming. As noted in [Lano and Haughton 96], the different machines can be related to each other through EXTENDS, INCLUDES, SEES, USES. However, the B language has some restrictions on the way different related machines can pass control to each other as follows:

- SEES can be applied in machines, refinements, or implementations. More than one machine can see a particular machine (shared access at all levels of development) that can be used for gathering the set of common types and constants.
- USES can only be used in machines. More than one machine can use a particular machine (shared access at specification level). The USES construct is usually used for shared access to readable data, like SEES, but within a subsystem development only.
- INCLUDES and EXTENDS can only be applied in machines, but not refinements nor implementations. A PROMOTES clause can be used to selectively promote operations from the included machine. The advantage of PROMOTES is important to enable in B, a machine cannot invoke operations unless they appear directly in the sub-machines defined in the INCLUDES part or a promoted machines from lower-levels.
- EXTENDS is the same as INCLUDES but with automatic promotion of all operations from the extended machine.
- Only one machine can include or extend a specific machine (exclusive access).

These restrictions require the system to be structured in a specific manner. Structuring is a step that has to be performed by the user (Software Engineer) to divide the code fragments or control sequences into different machines, some of the machines representing devices like actuators and sensors, and the rest representing the software controller for the system, which reads the input from the sensors, and then make appropriate decisions, and, finally, passes the decisions to the actuators to control the system. In large-scale systems, the controller itself needs to be broken down into a number of sub-controllers that together provide the whole function of the controller in a modular manner.

One of the key differences between B, on the one hand, and Z and VDM, on the other, is the way of changing the program state. In B, the substitutions defining the operations are meant to define these transitions. The generalised substitution approach simplifies the proof requirements and provides a uniform notation from abstract specification down to procedural code. Thus, these reasons made us choose to generate a B specification corresponding to the corrected requirements in GOPCSD.

4.5.2 B Toolkit

The B toolkit [B-core, Lano and Haughton 96] provides an IDE for developing, analysing, checking, refining, and implementing software specifications. The B toolkit has an auxiliary set of programs that are applied to the B language specifications. These auxiliary programs include many functions that can be summarised as follows:

- **Type Checker.** This tool detects type errors. In addition, it can detect errors related to inter-module visibility of both variables and constants.
- **Proof Obligation Generator.** It generates the proof obligations associated with a module. It generates non-trivial proofs, which are used to prove the initialisation and operation segments of the machines will obey the invariants.
- **Automatic prover.** It analyses the proof obligation and consults the mathematical rule base to simplify and discharge the proof obligation.
- **Interactive prover.** It allows the user to investigate the reasons for the failure of the proof. It traces all of the mathematical rules that have been applied during the proof. This tool ensures that the system requirements and environment rules prove the specifications. It requests the user to enter a list of rules about the environment to prove the satisfaction of the system properties within the specifications.
- **ADA and C translator.** This tool translates the specifications of B into high-level programming languages of Ada or C.
- **Latex Document Generator.** This tool produces documents related to the design of the application in Latex format.
- **Cross-reference** This tool highlights the sites where each variable is declared, used, or modified.
- **Dependence Graph Generator.** This tool determines which other modules a given module depends on.

4.6 Using B to Specify Reactive Systems

The B method can be applied in reactive systems development. Most reactive systems have safety and security concerns apart from operational ones, which entails a need for a formal model that will be used to generate an implementation conforming to the design. The B method can be used to specify reactive systems because it provides formal notations suitable to express the basic operations of reactive systems. Furthermore, the component-based nature of B can be used to represent system components, provided that the Systems Engineer models each

component as a separate specification machine. The B language provides standard relationships between the different machines of the developed system, like SEES, USES, INCLUDES and EXTENDS. These relationships enable the passing of control between the different components, which can be used to build hierarchical systems from the sub-components.

This hierarchical orientation of B machines encourages building reactive systems as a number of machines representing the sensors, controller(s) and actuators. Because the input data sensed by the different sensors of the system should be collected for the controller to take corresponding actions, a collection criterion is required. This criterion tests the different sensors in a specific order for new values, and then delivers the new values to the controller. Because the required collection operation is always the same for a given system, a separate machine, usually named the Interface Controller (IC), can be created and supplies the main controller with the newly sensed data to invoke the appropriate operation within the main controller. The main controller consists of the operations that control the different actuators, either directly in case of small-scale applications or via an appropriate structure of sub-controllers. The sensors can be represented as separate machines or included within the interface controller. Figure 4.4 has two Data Control Flow Diagrams (DCFD): the upper one represents a DCFD of a reactive system, while the lower one shows how to split the controller into a sensor interface and a main controller, as noted in [Lano et al. Y2Kb].

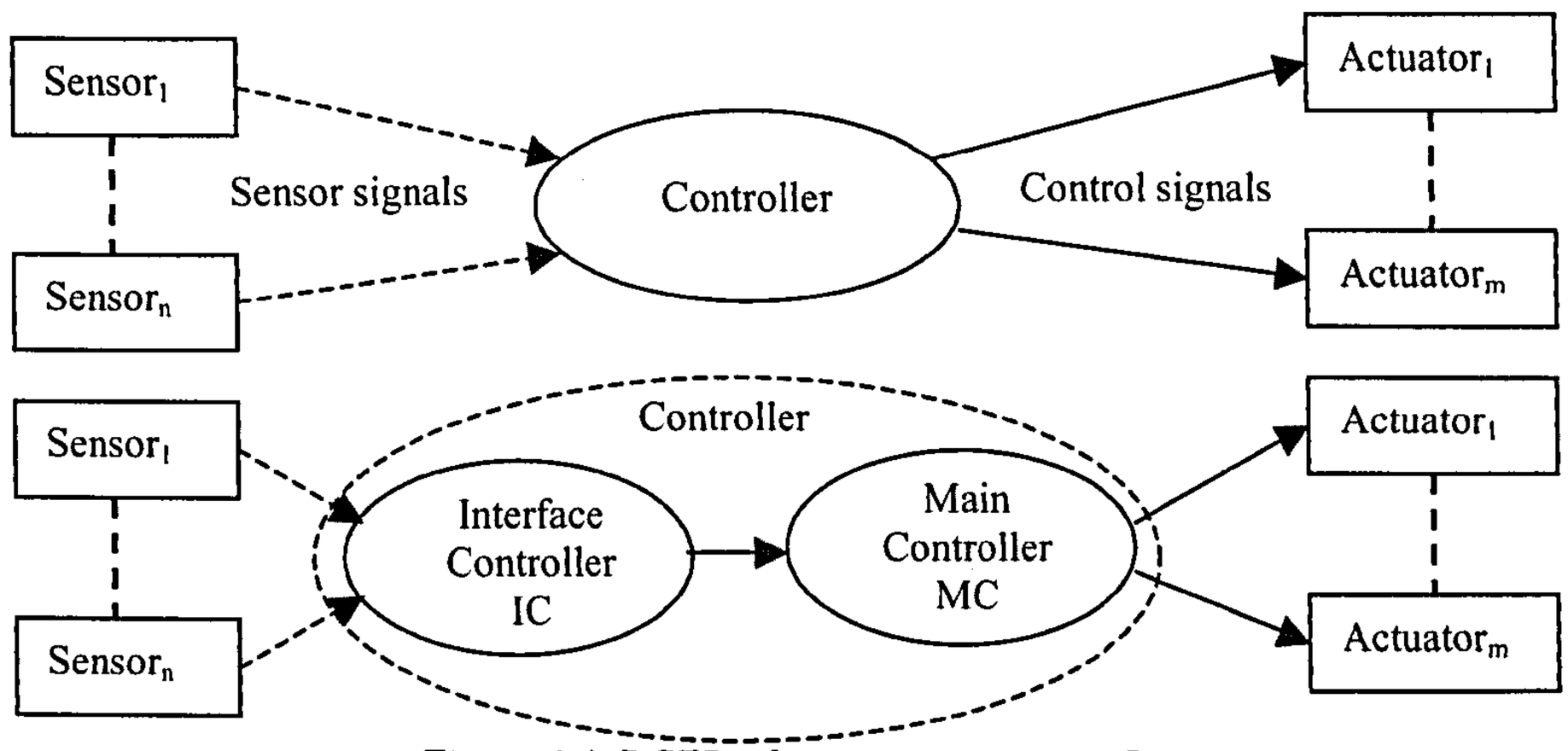


Figure 4.4, DCFD of a reactive system in B

4.7 Gas Burner System (an example of reactive system in B)

We examine an example of a gas burner system to illustrate how to use the B language to specify software controllers of reactive systems (this example follows the main steps of the RSDS method [Lano et al. Y2Kb]). The gas burner system is considered a reactive system as it has sensors that recognise the existence of the flame and the status of the switch. The gas burner has actuators to control the gas and air valves and to switch the igniter on and off.

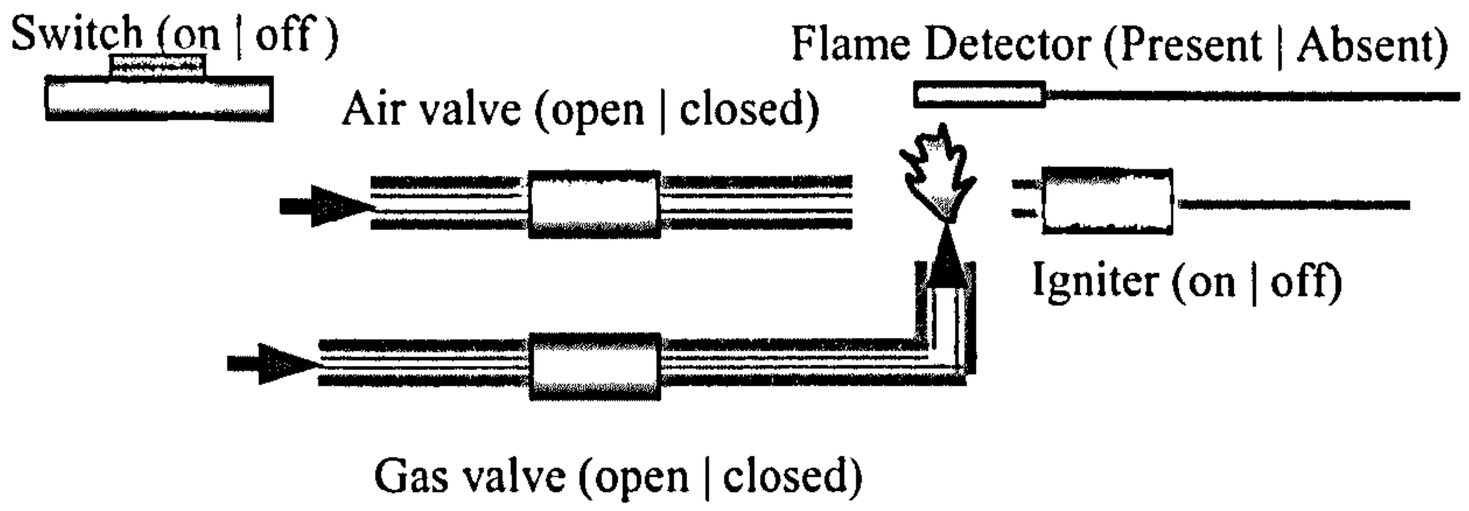


Figure 4.5, Gas Burner System

Figure 4.5 shows the basic gas burner system. The process of developing the specifications is similar to develop software applications; thus, we will not describe the details and the different approaches of the development but focus on the system, variables, events, transitions, and machines states.

4.7.1 The Problem Domain

As described in figure 4.2, the first step is to acquire a description of the problem domain and the informal system requirements. This describes the existing system components and provides a general perspective on the required software functions. Such specifications should clearly indicate the sequence of required operations for each expected event that can be sensed by the developed system. The operations should indicate what is the system's corresponding outputs or changes in the system's internal state according to the different events. Although the informal specifications can achieve a high level of understandability, the chance for deriving incomplete, ambiguous or inconsistent requirements specifications is high, due to the lack of a formal description. However, a basic form of specification for the gas burner system can be built on the basis of the following observations:

- Start up when required, when the user turns switch on.
- Shut down when required, when the user turns switch off.
- Maintain safety conditions; during operation avoid having the gas valve open while the air valve is closed.
- Maximise the lifetime of the igniter; do not switch on the igniter while the flame already exists.
- Keep the flame burning; keep the valves open to maintain the flame while the switch is on.

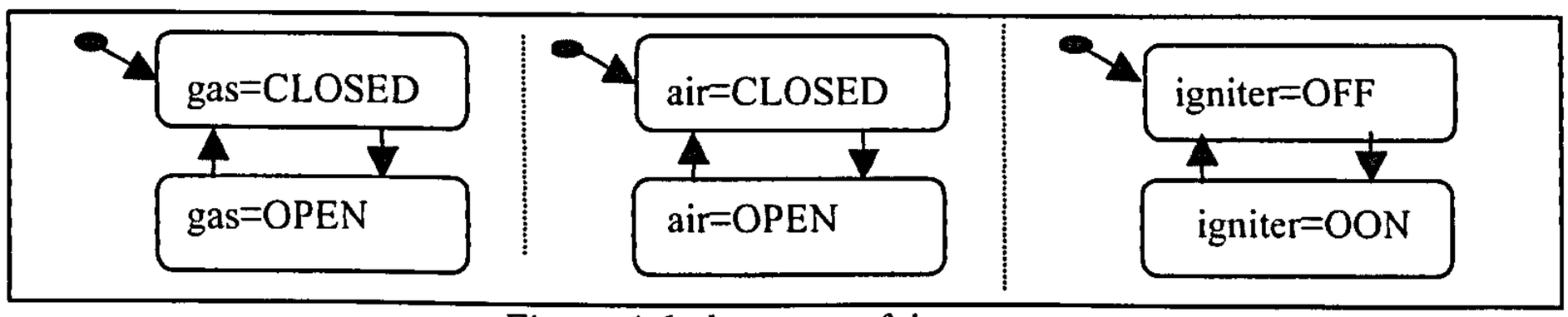


Figure 4.6, the states of the output

The gas burner system has five variables: two input variables (switch and flame detected) and three output variables (gas, air, igniter). The state description of the burner system can be represented by the output variables (gas, air and igniter), while the input

variables (or change of the input variables) can be used as stimuli for the different operations that change the system state.

To be able to derive early design for the gas burner, the different parts of the system have to be specified using a formal model. State Transition Diagrams/Statecharts are usually used because they can be straightforward translated to specification constructs in B (or other specification languages).

Figure 4.6 shows the state of the gas burner system as a combination of the three output variables gas, air and igniter. In figure 4.7, the different transitions between different states (combinations) of the input variables switch and flame_detected are shown with the appropriate operation to be activated by the developed application; in some combinations an action from is required even though the input variables remain the same. Tables 4.1 and 4.2 enumerate the different combinations and different transitions between these combinations; it should be noticed that some of these transitions are impossible; however, we show all of the combinations and transition for illustrative purposes; they can be used later in order to cover all of the odd cases that might occur during the operation of the developed application.

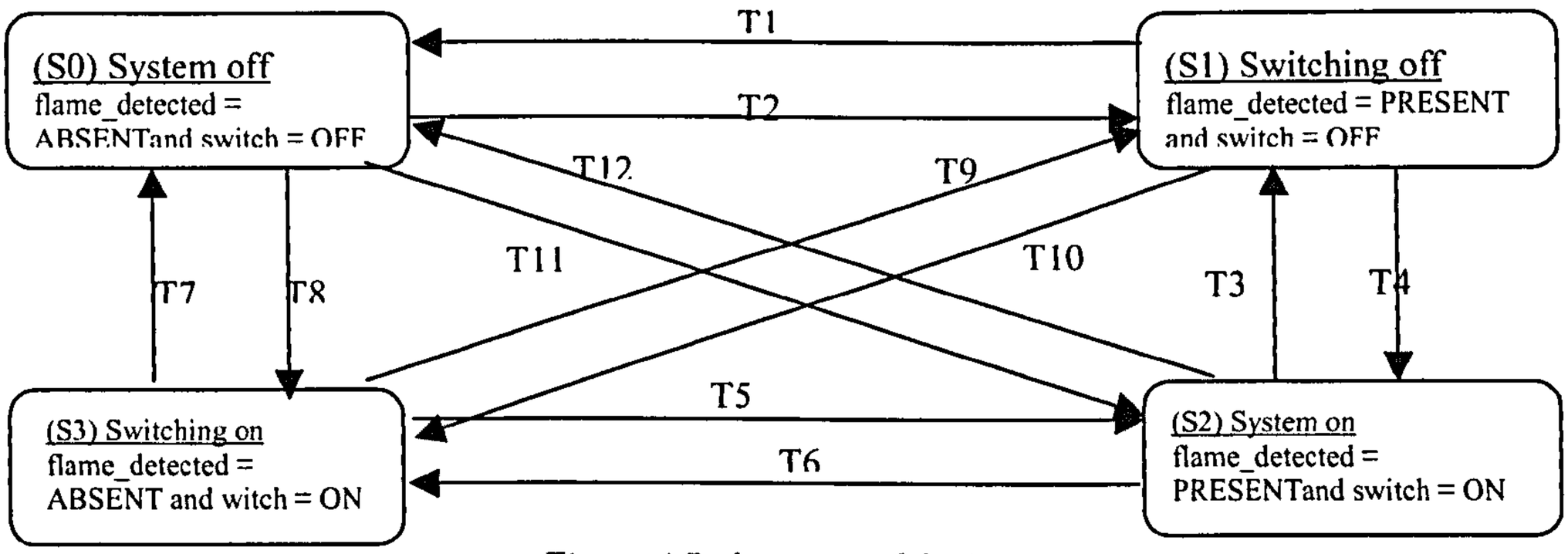


Figure 4.7, the states of the input

Table 4.1, the generated events for the transitions of the gas burner states

Tr	Description	Generated event
T1	The flame is present suddenly	Odd Case {Don't care}
T2	The flame disappears (after switching off)	No action {Don't care}
T3	Switch is switched on and flame is present	Odd Case {Don't care}
T4	The switch becomes off	Switch_off
T5	Flame disappears for some reason	Switch on
T6	Flame appears	Stop igniter
T7	Switch off while there was no flame	Odd Case {Don't care}
T8	Start up	Switch on
T9	Switch on and there is flame	Keep the flame
T10	Switch off suddenly	Switch off
T11	Flame appears and switch off	Switch off
T12	Flame disappears and switch on	Switch on

Table 4.2, the generated events for the states of the gas burner states

State	Description	Generated Event
S0	Switch is off and no flame detected	No action
S1	Switch is off but flame is detected	Switch off
S2	Switch is on and flame is detected	Keep the flame
S3	Switch is on but no flame detected	Switch on

4.7.2 The Gas Burner Specifications

As indicated earlier, the thread of control will start at the interface controller that examines the input variables and then invokes one of the main controller operations; the main control in turn invokes the actuators to change the overall state. When the local environment is affected by the actuators of the gas burner system, it may change the flame_detector_sensor reading or the gas_burner user may change the switch_sensor; in either cases, in the following cycle, the sensor changes will be responded to by the appropriate actuator(s). This activation will be continuously repeated during the operation of the gas burner system. Thus, the sensor machines can be included within the interface controller machine, while the actuator machines will be included within the main controller machine. The main controller itself can be included within the interface controller machine. This machine-structure follows the constraints of the RSDS tool and respects the uniqueness constraints of inclusion for the controller/actuator/sensor.

To avoid having to repeatedly redefine any numerated types more than once within these machines, a general machine for the data types can be used (burner data types) which can be related to each other machines through the see relationship. Figure 4.8 illustrates the relationship between the different machines of the gas burner system.

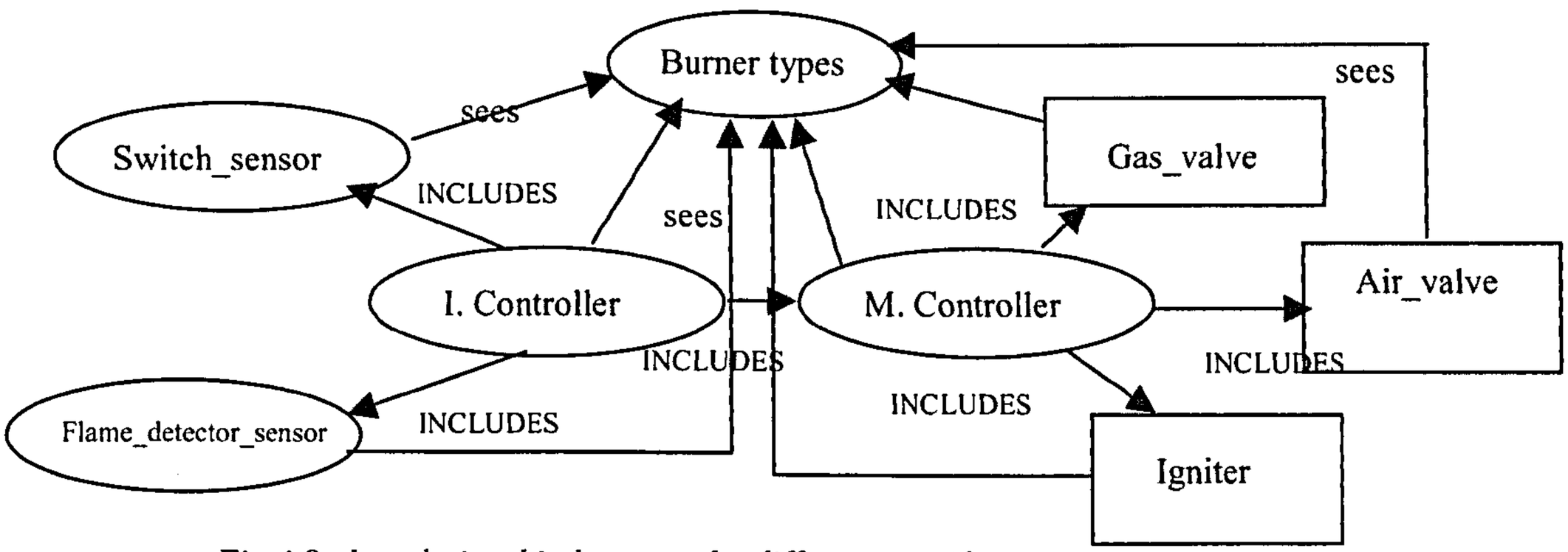


Fig 4.8, the relationship between the different specification pieces

It is important to mention that the approach used to build the specification in this section is not unique; other approaches can consider only the normal states and transitions of input variables, combine the interface controller and the main controller, combine the sensors within the interface controller, and/or have sub-controllers before the lowest level actuators. However, we illustrate the basic aspects of the specification design without going into deep details of alternative designs. According to figure 4.8, we list the specification machines required to build the gas burner application in the following sub-sections.

4.7.2.1 The B machine for the Gas burner data types

This machine is a global machine that will be seen by all other machines in the application and it has the definitions of all the user-defined types.

```
MACHINE Burner_Data_Types
```

```
      SETS
      VSTATES = {open, closed};    /* valve states for gas and
air valves*/
       ISTATES = {on, off};              /* Igniter states*/
       FDSTATES = {present, absent}     /* flame deetctor
states*/

END
```

4.7.2.2 The B machine for the Switch Sensor

This machine defines the input variable switch; and has an operation to read in the new reading from the switch hardware interface. The machine sees the global *Burner_Data_Types* machine and will be included within the interface controller machine to allow invoking of the *get_Switch_state* operation from within the interface main controller machine.

```
MACHINE Switch_sensor
      SEES Burner_Data_Types
      VARIABLES switch
      INVARIANT switch : ISTATES
      INITIALISATION switch := off

      OPERATION
       get_Switch_state = /* this operation will be replaced by
the  interface read function*/
      PRE true
      THEN
      ANY   xx
      WHERE  xx: ISTATES
        switch := xx
      END
END
```

4.7.2.3 The B machine for the Flame Detector Sensor

This machine defines the input variable flame detector; and has an operation to read in the new reading from the flame detector hardware interface. The machine sees the global *Burner_Data_Types* machine and will be included within the interface controller machine to allow invoking the *get_Flame_state* operation from within the interface main controller machine.

```
MACHINE flame_detector_sensor
      SEES Burner_Data_Types
      VARIABLES flame_detected
      INVARIANT flame_detected : FDSTATES
      INITIALISATION flame_detected := absent

      OPERATION
       get_Flame_state = /* this operation will be replaced by
the interface read function*/
      PRE
        true
      THEN
        ANY  xx
         WHERE  xx: FDSTATES
         flame_detected := xx
       END
 END
```

4.7.2.4 The B machine for the Interface Controller

This machine specifies the interface actions, which update the sensor reading by invoking reading operations from the sensor machines and comparing those readings with the old values to decide which operation within the main controller are to be invoked.

```
MACHINE Interface_Controller
      SEES Burner_Data_Types
      INCLUDES flame_detector_sensor, switch_sensor,
main_controller
      VARIBLAES old_switch, old_flame_detected
      INVARIANTS old_switch:ISTATES, old_flame_detected:
FDSTATES
      INITIAIALISATION old_switch, old_flame_detected := off,
absent

      OPERATIONS
        switch_Igniter_On=
        PRE
         true
          THEN
            get_Flame_state || get_Switch_state ||

          IF  /* invoke switch_on operation of MC machine */
             old_switch = off & old_flame_detected = absent
             & switch = on & flame_detected = absent THEN
switch_on
          ELSIF
          ...
         ELSIF /* invoke switch_on operation of the MC */
         old_switch = on & old_flame_detected = present
         & switch = off & flame_detected = present THEN
switch_off

        ...
      || old_switch:=switch || old_flame_detected :=
flame_detected

    END
END
```

Apart from the invariants concerning the variable types, the interface controller is usually where the invariants concerned with the input and output variables are specified; the input variables are already visible to the interface controller because it includes the sensor machines where the input variables are defined; and also the output variables are defined in the actuator machines, which are included within the main controller machine that is included in the interface controller machine.

The interface controller machine can vary dramatically from one implementation to another. The variation usually arises from the user intention to validate the controller by considering all-possible input combinations. Further, the controller needs to be interfaced with the sensor machines that may already exist.

4.7.2.5 The B machine for the Gas Valve Actuator

The air valve actuator machine defines the gas variable and contains two main operations to open and close the gas valve; this machine sees the *Burner_Data_Types* machine

and will be included within the main controller machine to allow invoking of the operations from within the main controller machine.

```
MACHINE gas_valve
      SEES Burner_Data_Types
      VARIBLAES gas
      INVARIANT gas: VSTATE
      INITIALISATION gas:=closed

      OPERATIONS
          open_Gas_valve=
            PRE
              gas=closed
            THEN
              gas:=open
           END;
          close_Gas_Valve=
              PRE
                gas=open
              THEN
                gas:=closed
            END
 END
```

4.7.2.6 The B machine for the Air Valve Actuator

The air valve actuator machine defines the air variable and contains two main operations to open and close the air valve; this machine sees the *Burner_Data_Types* machine and will be included within the main controller machine to allow invocation of the operations from within the main controller machine.

```
MACHINE air_valve
      SEES Burner_Data_Types
      VARIBLAES air
      INVARIANTS air: VSTATES
      INITIALISATION air:=closed

      OPERATIONS
          open_Air_valve=
            PRE
              air=closed
            THEN
              air:=closed
           END;
          close_Air_Valve=
            PRE
              air=open
            THEN
              air:=closed
           END
 END
```

4.7.2.7 The B machine for the Igniter actuator

The air valve actuator machine defines the igniter variable and contains two main operations to switch on and off the igniter; this machine sees the *Burner_Data_Types* machine and will be included within the main controller machine to allow invocation of the operations from within the main controller machine.

```
MACHINE igniter
      SEES Burner_Data_Types
```

```
        VARIBLAES igniter
        INVARIANTS igniter: ISTATE
        INITIALISATION igniter:= off
        OPERATIONS
             switch_Igniter_On=
              PRE
                igniter=off
              THEN
                 igniter:=on
              END;
             switch_Igniter_Off=
               PRE
                 igniter =on
               THEN
                 igniter:=off
               END
END
```

4.7.2.8 The B machine for the main controller

This machine controls the other actuators; its operations are called from within the interface controller machine in response to sensor changes. This main controller machine sees the burner_data_types!!!!!!! and includes the actuator machines. Besides the operations, the main controller machine should include the invariants governing the application, like the safety invariant concerning the necessity to have the air valve open when the gas valve is open. The invariants should be separate from the operations and, furthermore, they should be preserved by the operations and the initialisation of the main controller machine and other related machines.

```
MACHINE main_Controller
        SEES Burner_Data_Types
        INCLUDES air_valve, gas_valve, igniter
        INVARIANTS gas=open=> air=open & ...

        OPERATIONS
          start_up =
           PRE
             air=closed & gas =closed & igniter=off
           THEN
            open_Air_valve || open_Gas_valve ||switch_Ingniter_on
           END;
         shut_down=
          PRE
             air=open & gas =open
           THEN
             close_Air_valve || close_Gas_valve
          END;
        ...
        Maximize_IgniterLT=
          PRE
           igniter=on
         THEN
            switch_Inginter_off
          END
END
```

4.7.3 Structuring the specifications

In case of large-scale applications with many separate components, it is usually preferable to structure the controller into small-size modular sub-controllers; this approach

enables easier implementation and maintenance, as well as checking and debugging within the B toolkit environment.

The B language has some enforced restrictions on the manner of extending and or including machines within other machines. These restrictions prohibit including the same machine more than once; this ensures uniqueness of control. Thus, the sub controllers need to be structured in a way that preserves the B constraints. The different structures can be constructed through considering the system invariants that take the following form, as in 4.1:

$$F(Var_1, Var_2, Var_3 \ldots, Var_n) \Rightarrow G(Var_m, \ldots, Var_k) \quad \textbf{(4.1)}$$

where variable Var_i is one of the application's components' variables and *G* and *F* are two Boolean functions combining different combinations of the components' variables. For example, the safety part of requirements given in section 4.6.1 relates the two variables *gas* and *air (*and hence the two components air valve and gas valve) through the invariant 4.2:

$$gas = open \Rightarrow air = open \quad \textbf{(4.2)}$$

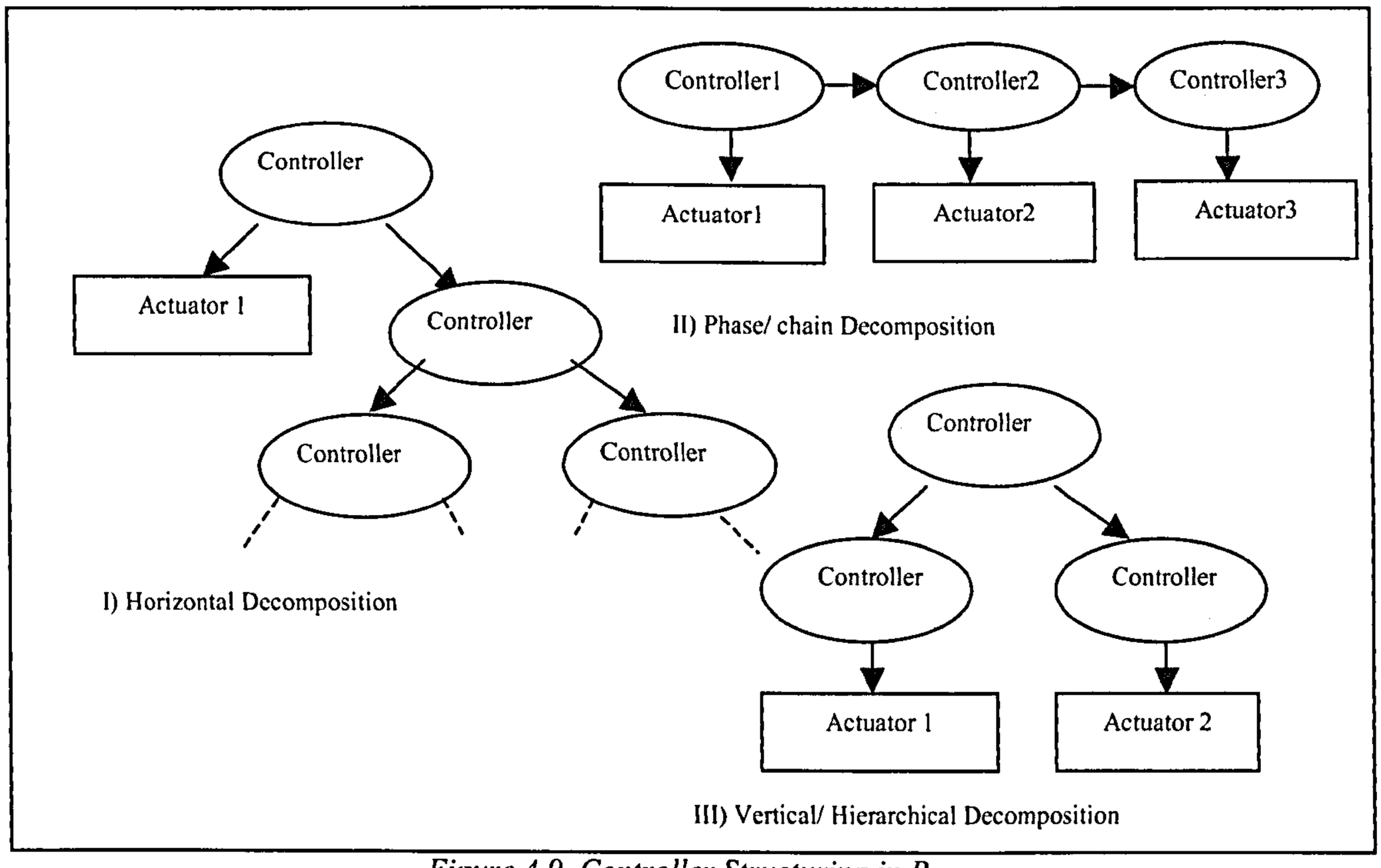


Figure 4.9, Controller Structuring in B

Lano et al. in [Lano et al. Y2Kb] proposed a method based on the system invariants defined by the Systems Engineer to structure the controller in conformance to the B constraints. Looking at the different patterns of the variables appearing in both sides of 4.1, they provided the following guidelines:

As noted in [Lano et al. Y2Kb], when there is total independence between the sub-components; i.e. none of the components depends on the internal state of the others, the horizontal structuring can be used. But, when there is relative independence of the control invariants associated with the individual sub-components, the hierarchical control structure can be used. In which case, the interaction between the sub components is managed by a supervisor controller. In other cases, a different structuring mechanism, called a chain, may be required,

especially when the control action or decision can be divided among sub-controllers each of which do a specific job; for example, checking faults and controlling the plant - in case of an existing fault, the control will not be passed any further. Figure 4.10 [Lano et al. Y2Kb] shows these three types of structure.

It is important to mention that some other type of structure, like temporal structure in which the same actuator or sub-controller appears more than once under different super-controllers, cannot be achieved within B, because of structuring rules, unless the super controllers are combined together into one machine.

4.7.4 Specification Refinement

The B language allows users to refine the abstract level of coding within the specification machines. The operations that were defined inside the B machine will be refined within the refined machine. Refining the operations will involve imposing more determinism in assigning values to variables. Adding new variables to describe the machine more precisely is a common step in refinement. Figure 4.10 [Lano and Haughton 96] shows a specification machine in B (M to the left) and a refinement of it to the right (R); the machine is deemed to be

```
MACHINE
        M
VARIABLES
        V
INVARIANTS
        INV
INITIALIASATION
        INT
OEPRATIONS
        Operation1=
        PRE pcond
                THEN
                  S
        END;
...
END
```

```
REFINEMENT
        R
REFINES
        M
CONSTRAINTS
        CSTr
VARIABLES
        Vr
INVARIANTS
        INVr
INITIALIASATION
        INTr
OEPRATIONS
        Operation1=
        PRE pcondr
                THEN
                  Sr
        END;
...
END
```

a refinement if it has a section REFINES that indicates which specification machine it refines.

Figure 4.10, machine refinement in B

The refinement R will be considered a correct refinement of the machine M when R satisfies some conditions on its states, initialisation, and operations. Such conditions are called proof obligations and are listed as stated in [Lano and Haughton 96]:

- *There is a combined abstract and concrete state that satisfies the refinement relation and the abstract machine invariant.*
- *The concrete initialisation (INTr) is a refinement of the abstract machine initialisation (INT), under the assumptions of the constraints and properties of both machines (R and M).*
- *Under the refinement relation and the precondition of the more abstract operation (pcond), the precondition of the more concrete operation holds and for every execution of S there is a corresponding execution of (Sr) from the same initial state*

that re-establishes the refinement relation between the following states of the abstract and concrete machines.

4.7.5 Implementation

The refinement operation can be repeated for each machine until all required details have been expressed. In this respect, the implementation is considered as the final refinement because it refines the last version of the specification. The implementation version usually includes input and output libraries to enable monitoring information to the end user and receiving inputs from him in other situations. Figure 4.11 [Lano and Haughton 96] shows that a machine M can be refined to a refinement R1 followed by refining the refinement R1 to R2 FIGURE PLACEMENT PROBLEM ABOVE?until reaching a final refinement machine that is called the implementation (I). The relevant proof obligations must be satisfied between (M and R1), (R1 and R2), ..., and (Rn and I).

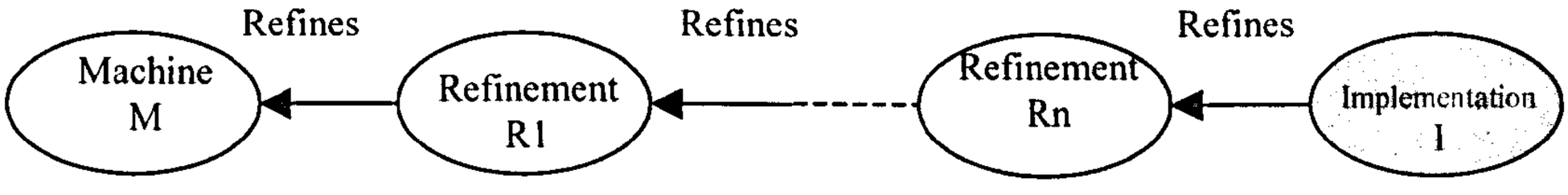


Figure 4.11, Successive Refinements in B

4.8 Conclusions

Formal methods play an important role in the development lifecycle of software applications; they provide the user with mathematical and logical notations to enable him/her to express and reason about the main decisions in building the applications. There are algebraic specification languages such as OBJ and model-oriented formal methods like Z, VDM, and B.

The B method has an integrated development environment, the B toolkit, which helps in refining, animating and proving the formal specifications. The B method has a component-based nature that enables expressing a large application in terms of hierarchies of specification units (machines); this fits well with the nature of process-control applications. The requirements of the reactive systems can be formally specified in the form of B machines; and, further processed within the B toolkit environment. The specification operation requires co-operation between a systems engineer and a software engineer who translates the requirements into B formal specifications.

The GOPCSD Method 5

In this chapter, we describe the requirements analysis method supported by GOPCSD: we discuss the method's objectives, the adaptations of the KAOS method to fit efficiently with process control applications, the goal refinement patterns, the method's support for reusability, and the semantics of goal-refinement.

5.1 The outline of the GOPCSD method

As indicated earlier in chapter 2, section 2.3, one of the crucial objectives of the GOPCSD method is to achieve the separation of concerns between the process control systems engineers and the software engineers. In [El-Maddah and Maibaum 2003a], we have established the main phases required to specify process control applications and reason about their requirements, before translating them into B formal specifications. In figure 5.1, we sketch the internal structure of the GOPCSD tool, built to demonstrate the method. To the left hand side, we sketch the requirements development lifecycle.

The GOPCSD method covers the early stages of the development lifecycle for process control systems. It addresses defining abstract requirements, refining and formalising the user needs and finally checking the requirements and compiling them to formal specification forms. The interaction between the process control systems engineer and GOPCSD covers phases one and two, where, in the first phase, the tool supporting the method should provide a library of requirements segments to be imported into the application; and enabling the process control systems engineer to use a supporting tool to construct goal-models from scratch in case they are not provided in the library. Then, the systems engineer (the user) combines these requirements segments into a complete goal-model. The

second phase uses checks and validation to enable judging the suitability of the constructed goal-model. For this reason, the process control systems engineer will have a chance to debug and modify the requirements, before advancing to the final phase, where an automatic translation will generate B specifications from the requirements goal-model. A software engineer will later process the generated B specification within the B toolkit environment.

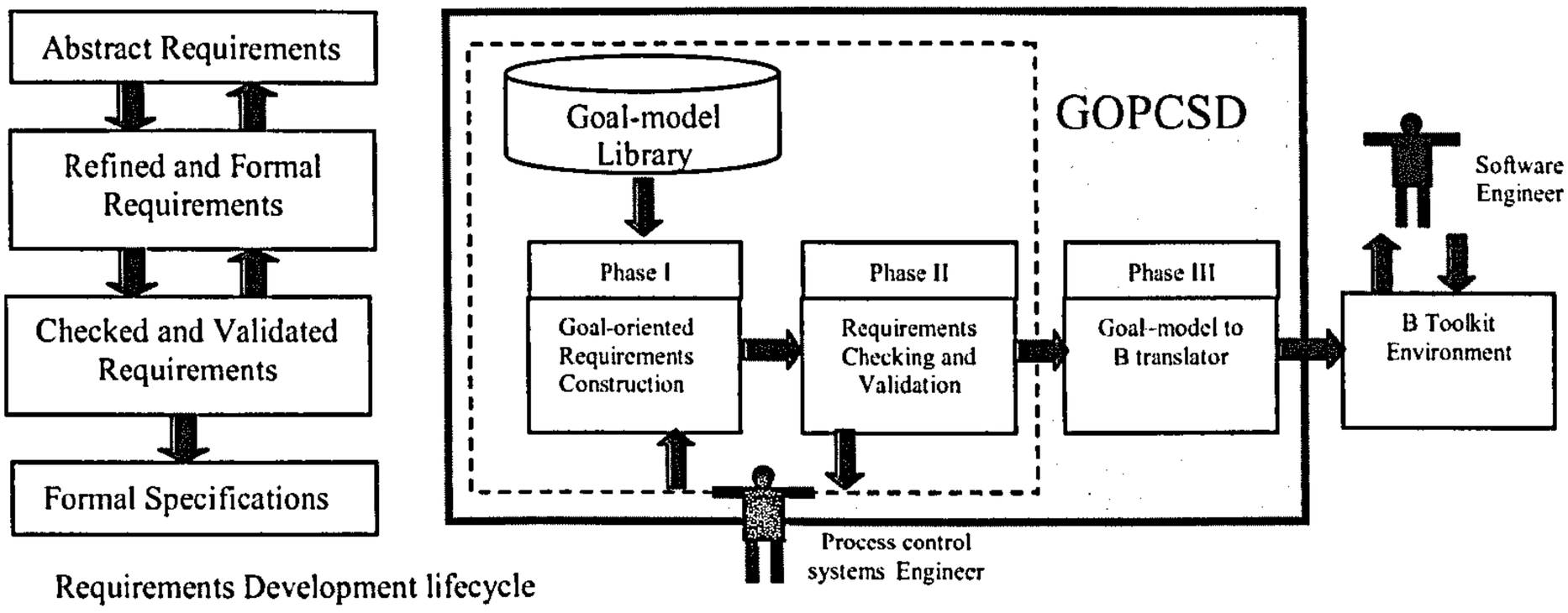


Figure 5.1,GOPCSD structure and the related development lifecycle

Unlike the first and second phases, the third phase is hidden from the user in order to increase the feasibility of using the method and make it independent of the formal method. In other words, an ordinary Process control systems engineer who does not know the details of programming languages or formal methods can use the method and its supporting tool efficiently and reason about the requirements. Although the first two phases of the method are based on the corresponding ones in the KAOS method, we considerably adapted the KAOS method to suit the nature of process control systems; on the one hand, we refined some abstracted steps in the KAOS method, which is usefully defined in detail for process-control applications; on the other hand, we briefly describe other steps, which mainly deal with identifying the main components and goals of the application. As we introduce a reusable goal-models library, we integrate the importing and mapping of goal-models within the first phase where we construct the goal-model. In addition to the adaptation in the first two phases, we append a third phase to generate the formal specifications by translating the requirements goal-models to B machines.

5.2 Adapting the KAOS method

Process control systems are composed of easily identifiable (physical) components. Their operation can be basically specified using sequential and simultaneous operations. However, since KAOS is a general method for specifying software applications, it does not take into consideration these mentioned features of process control systems. Thus, we introduced some modifications to the data models and the algorithms of the KAOS method so that our tool can effectively deal with these features.

5.2.1 The Object-Model

The KAOS method adaptations start at the object-model level, which provides a formal description of the application. However, with respect to the nature of process control systems, the

elements of the KAOS object-model can be represented within the GOPCSD tool component, variable, agent and goal-model lists. The relationship between the application and these elements will be implied from the lists, but not separately represented as class or entity relationship diagrams.

5.2.2 The Goal-Model

Since patterns are the key point of reusability, identifying patterns at the level of Requirements, Design, or even Implementation reduces the time and the effort required to produce similar applications. The goal-models of the KAOS method are represented by AND-OR trees. Thus, there are two refinement patterns in KAOS: AND and OR. The OR refinement has the meaning of alternative solution: each sub-goal can represent a different way of achieving the main goal, while the AND means conjunction; each conjunct sub-goal is required to achieve the main goal.

Refinement patterns provide considerable guidance to the user in reasoning about and debugging the constructed requirements since they are closer to his/her perspective and address rational concepts like sequentiality, alternation and simultaneity. Thus, we examined different process control case studies (a tank system, production cell, simple gas burner system, and a lift system) and identified six recurring patterns: two of them extend the existing patterns in the KAOS method (alternative and conjunction) and four are new (sequence, disjunction, simultaneous and inheritance) representing refinements of the AND pattern. The hierarchy of the requirements goals that are related to each other through refinement patterns will enable the user to derive the outlines of the implementation programs. This usually happens with less effort from the process control systems engineer as well as requiring less expertise in programming language paradigms and control structures. Further, these refinement patterns guide the translation to B machines, as will be illustrated in section 5.3.

5.2.3 The Agent-Model

The general nature of the KAOS method allows the agents to have various types: devices, humans or software components. The same range of agents is available in the GOPCSD tool; however, the agents can be assigned to accomplish a terminal goal even if the goal's formal condition contains some variables that are not observable by the agent. The difference in the agent treatment in the GOPCSD method arises because the formal description of the goal can be regarded as a condition plus an action: the condition can be checked in one module representing the controller while the action can be activated in another module representing the actuator device.

5.2.4 The Operational-Model

In Process Control Systems, the operations required to accomplish the terminal goals can be represented as conditional assignments for the output variables in the form: *condition* $\Rightarrow$ *variable* = *value* where *condition* is a Boolean expression composed of simple comparisons between the application variables and their corresponding values, *variable* is one of the controllable output variables and value is a defined value belonging to its domain.

Thus, the operation-model can be combined with the formal description of the terminal goals instead of being described separately.

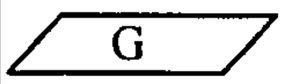
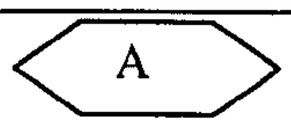
5.2.5 Components-first Development

As process control systems are constructed from existing components, we adopt a component-base development method, in which the user can reuse already defined components' goal-models and refine the overall system goals into these components' sub-goals. Thus, instead of freely extending the goals in KAOS to explore the existing system and environment, in GOPCSD, the system goals will be refined into target sub-goals. Although, the component sub-goals are known in advance, their arrangement is unknown and the arrangement needs to be checked and validated to specify the system correctly.

5.3 The Refinement Patterns

In this section we describe the meaning of each identified goal-refinement pattern. Further, we provide general description of situations where the pattern can be employed. Table 5.1 shows the graphical symbols that will be used to express goal-models.

Table 5.1, the goal-model graphical symbols used in the thesis and the tool.

The element	Symbol	Comments
Alternative		A separate line for each sub-goal
Sequence		An arrow pointing from the first sub-goal towards the last sub-goal (usually, the arrow will be drawn from left to right)
Conjunction		A double arc over the sub-goal lines indicating a conjunction refinement site.
Disjunction		A single arc over the sub-goal lines indicating a disjunction refinement site.
Simultaneous		A single arc over the sub-goal lines with single lines between each two adjacent goals
Inheritance		A single bold arrow pointing from the child-goal to the parent goal (similar to "is an instance of" relationship within UML and ERD)
Goal		A trapezoid to represent goals that have some detail like name or index.
Goals without details		A circle to represent goals without any details
Agent		A hexagon to represent agents
Accomplish		A curved line to represent the relationship between an agent and a goal.

5.3.1 The Alternative Refinement Pattern

The alternative refinement pattern is the KAOS OR pattern. This pattern means each of the sub-goals can be used separately to achieve the main goal. Accordingly, the alternative pattern should be used when there is a preference to express the availability of different solutions within a single goal-model. For example, to achieve one goal for shutting down a gas burner system, and according to safety consideration, the shut down goal can be refined as an alterative of two sub-goals: the first goal

(the gas valve will be closed first and then the air valve) and the second goal (the two valves will be closed simultaneously). At a later development stage, each of these two sub-goals will appear in a separate final version of the goal-model.

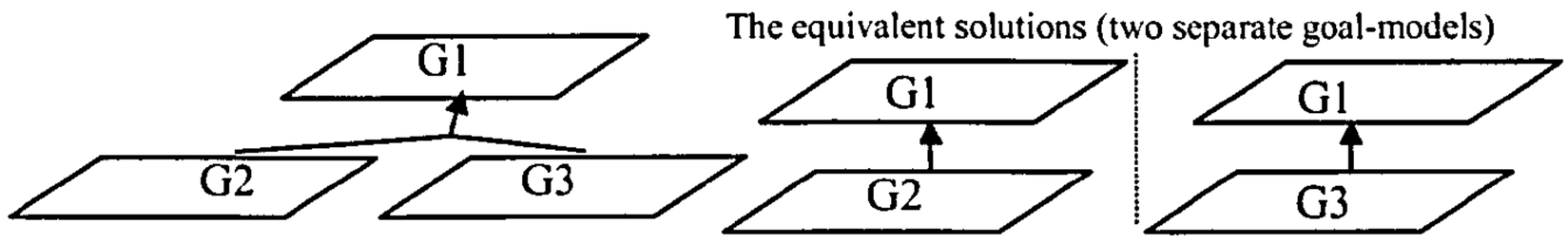


Figure 5.2, the alternative pattern

In figure 5.2 we sketch a goal-model template (incomplete) to the left, where the non-terminal goal G1 is refined as the alternative of the two goals G2 and G3. This fact can be expressed notationally, as *G1=Alternative(G2,G3).* To the right of the goal-model template, a set of two goal-models is generated as a result of splitting the goal-model to the left at the alternative refinement site. Each sub-goal G2 and G3 appears separately in one of the two goal-models and the refinement relationship in the new goal-model will be *Inheritance* instead of *Alternative.*

5.3.2 The Sequence Refinement Pattern

The sequence pattern is frequently used in process control systems. It is used to express a higher task in terms of a number of ordered steps; when each of these steps is achieved in order, the higher task will be achieved.

For example, a goal of delivering processed metals in a production cell application can be refined as a sequence of three goals: the first goal, feeding the blank metal, the second goal, processing the blank metal, and finally the third goal, delivering the processed metals. As indicated in this example, each step relies on the outcomes of the previous steps. For instance, the blank metal cannot be processed unless it has been fed to the press. In addition, we may notice the sub-goals themselves may need further refinement stages, as feeding the blank metals may need more refined steps to be achieved. Thus, when a goal is refined to a sequence of sub-goals, each sub-goal's accomplishment is considered as a pre-condition for its successor goal.

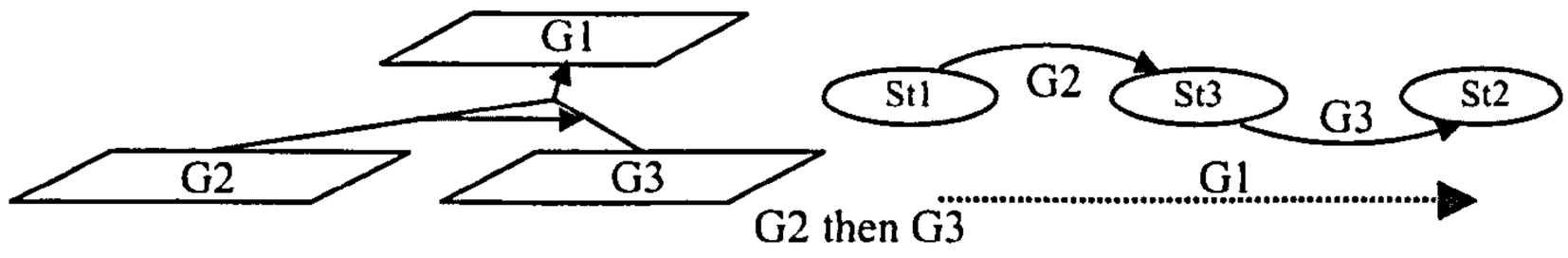


Figure 5.3, the sequence pattern

In figure 5.3, the goal-model template to the left expresses that goal G1 is refined as the sequence of goals G2 and G3, respectively. We use this notation: *G1= Sequence(G2,G3)* to express the sequence refinement. There is an equivalent state transition diagram (STD) to the goal-model. Each goal in the goal-model is represented as a transition in the STD from one state corresponding to goal's pre-condition to another state corresponding to the goal's post-condition. This analogy shows that the intermediate state of the STD (*St3)* represents the pre-condition of goal G3 and at the same time represents the post-condition of goal G2. Therefore, within the sequence refinement pattern, the post-

conditions of the predecessor goals must be considered. This will be formally discussed later in section 5.5.

5.3.3 The Conjunction Refinement Pattern

Conjunction patterns are usually used to refine the main goal into two or more sub-goals that need to be conjunctively accomplished. If the sub-goals do not possess any sequence or order dependency between each other, we can express the sub-goals as conjunction refinement of the parent goal. The main goal will be considered achieved only when each of its sub-goals is achieved. This refinement pattern is more likely to exist at the high-level segments of the goal-model where the sub-goals themselves are non-terminal and need further refinements.

For example, when designing a process control application, one should consider the different aspects like safety, running cost, security, and operation. These aspects can be represented through high-level goals that can be refined to narrower sub-goals. Thus, the issue of the conjunctive achievement of the different aspects can be represented as a conjunction between these high-level goals. For example, in a gas burner system, the ignition trials can be prohibited if there is a detected flame; this aspect can reduce the running cost of the system by increasing the lifetime of the igniter and reducing the consumed power. Thus, the main goal of the gas burner can possibly be refined into a conjunction of the operation goal and the cost reduction goal.

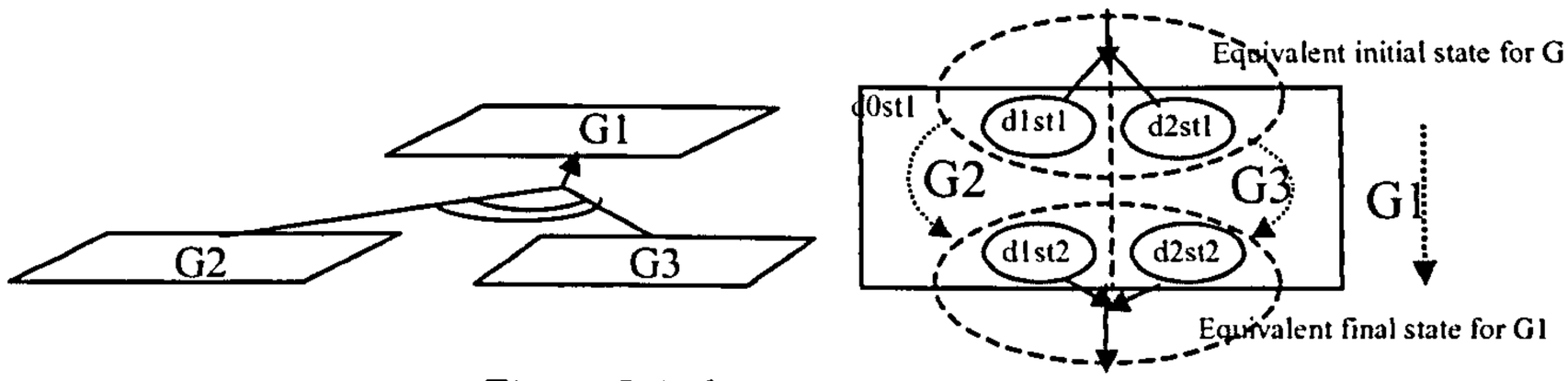


Figure 5.4, the conjunction pattern

In figure 5.4, goal G1 is refined as the conjunction of goals G2 and G3. This can be expressed notationally as *G1=Conjunction(G2,G3)*. To the right hand side, there is a sketch of an equivalent STD that has two sub-spaces, one for each conjunct sub-goal. This compound STD illustrates that the pre-conditions of goals G2 and G3 should imply the pre-condition of goal G1. In addition, having achieved the post-condition of goals G2 and G3 should guarantee the achievement of the post-condition of goal G1.

5.3.4 The Disjunction Refinement Pattern

This pattern has some features of the KAOS OR and AND patterns. It resembles the alternative pattern from the aspect that the parent goal can be achieved by achieving one of the sub-goals. On the other hand, the different sub-goals accomplish the parent goal in different situations; thus, removing one of them means that the parent goal will not be achieved in some circumstance. This pattern is repeatedly found in process control systems that achieve tasks differently from one situation to another.

For example, a double press production cell can stamp the blank metals in one of the presses according to the availability of the presses; thus, a parent goal to stamp metals can be refined to two sub-goals: the first is to stamp the metal in press one under some condition and the second goal is to

stamp the metal in press two under some other condition, but guaranteeing that the first condition has not been met. This shows that the two sub-goals are disjoint.

Another situation to use this refinement pattern is when grouping exclusive functions together; for example in a gas burner system, the basic functions of starting up and shutting down are considered exclusive because neither of them can operate at the same time as the other one. This exclusion ensures the pre-conditions will be disjoint, and hence, we can formulate a higher goal, named fulfil requests, which is refined to the disjunction of two sub-goals: shut down and start up.

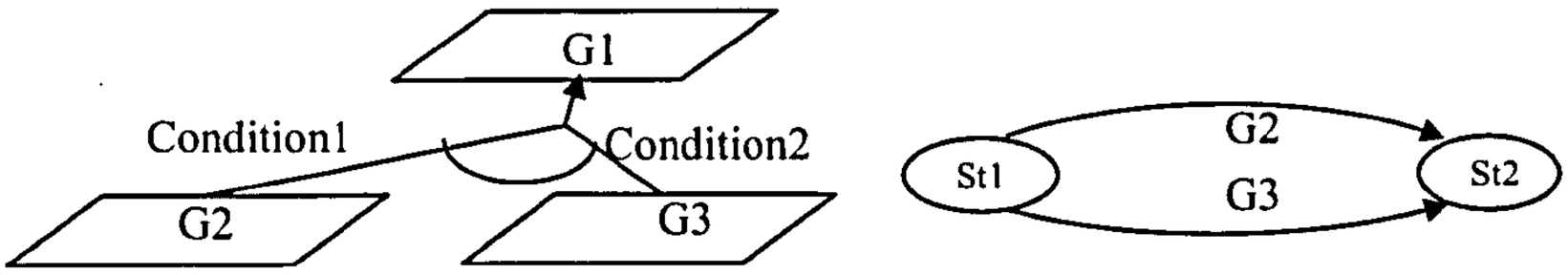


Figure 5.5, the disjunction pattern

In figure 5.5, goal G1 is refined as the disjunction of goals G2 and G3. This can be expressed notationally as *G1=Disjunction(G2,G3).* To the right there is a sketch of a STD that has two transitions, one for each sub-goal. Furthermore, we can notice that the pre-condition of G2 added to - logical or ∨- the pre-condition of goal G3 implies the pre-condition of goal G1. Or we can state that goal G1 is active when any of goals G3 and G2 is active. The post-condition of goals G2, G3 and G1 should be the same.

5.3.5 The Simultaneous Refinement Pattern

The simultaneous pattern is commonly required within process control systems. The parent goal will be refined to a number of simultaneously operating goals that may control different parts of the application.

For instance, in chemical applications, some situations can arise where pouring two chemical compounds or adding a substance and stirring the mixture are required to commence at the same time.

Although to certain extent the simultaneous pattern is similar to the conjunction pattern, they differ in the restriction for starting the actions. This difference delays the appearance of the simultaneous pattern to one level above the terminal-goal level, where the sub-goals of the simultaneous pattern will be terminal goals. This powerfully employs the simultaneous pattern to synchronise the operation of sub-components of the application.

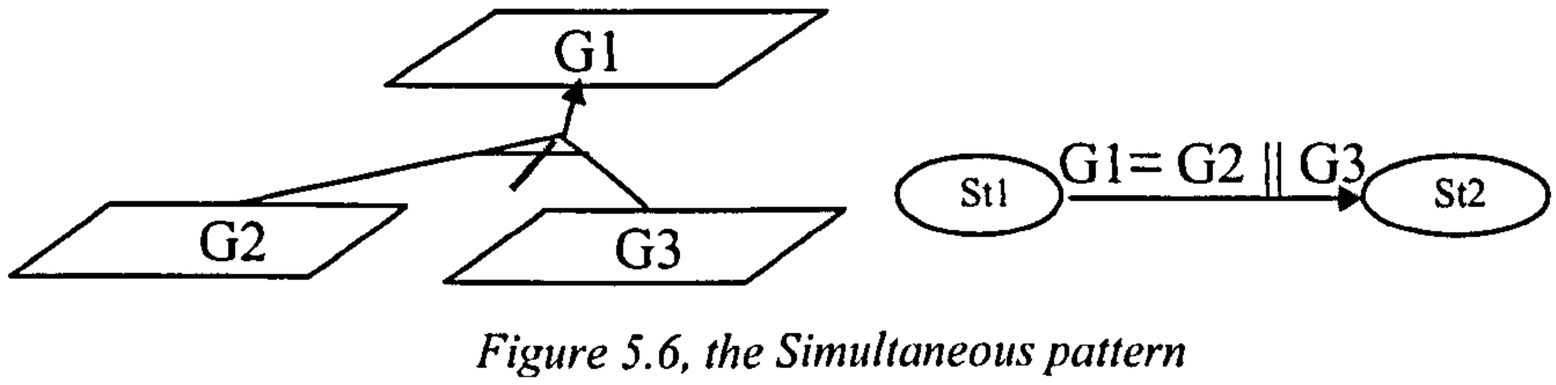


Figure 5.6, the Simultaneous pattern

In figure 5.6, goal G1 is defined as the simultaneous refinement of goals G2 and G3. This can be expressed notationally as *G1=Simultaneous(G2, G3).* The equivalent STD to the right from the goal-model shows that the post-condition of goal G2 added to, logically ∧, the post-condition of goal

G3 implies the post-condition of goal G1. Moreover, the pre-conditions of G2 or G3 should imply the pre-condition of goal G1.

5.3.6 The Inheritance Refinement Pattern

The inheritance pattern is a special case of refinement patterns where there is only one single sub-goal that has the same formal description as the main goal.

There are two main motivations for defining this pattern: first, when the main goal is too general to be refined directly, there will be a need for an intermediate goal that is stronger and/or more specific than the main goal. For example, in a lift system, defining a goal like maximising the lifetime of the cabinet motor would be too general to be refined. But, a goal like avoiding unwanted direction change could bridge the big distance between the level of motor and requests, on the one hand, and the high-level issues of motor lifetime, on the other. The second motivation is to maintain the refinement level after splitting the compound goal-models, where inheritance sites will replace the alternative sites in the new simple goal models, as shown in figure 5.2. This replacement fills the large refinement distance between goals G2 or G3, on the one hand, and the parent goal of goal G1, on the other.

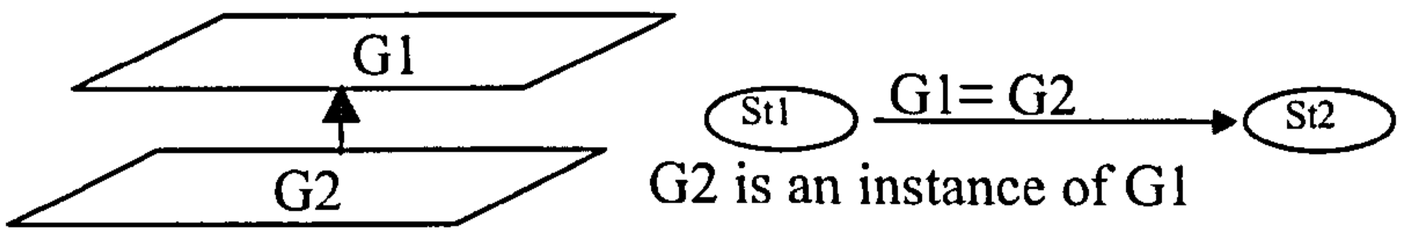


Figure 5.7, the Inheritance pattern

In figure 5.7, goal G2 inherits goal G1; this can be notationally described as *G1=Inheritance(G2)*. To allow the child goal G2 to be stronger than parent goal (i.e. more restricted pre-condition), the pre-condition of the child goal G2 should imply the pre-condition of the parent goal G1; while the post-condition of the parent should imply the post-condition of the child goal. To avoid any circumstance of ambiguity and inconsistency, we restricted the inheritance refinement sub-goals to be only one sub-goal per parent goal per goal-model otherwise, some situations may arise where there will be need for a selection criteria to choose which sub-goal to be activated when the parent goal is activated.

5.4 The GOPCSD Support for Reusability

Reusability concepts are usually employed within software engineering to reduce the time and effort required to develop similar applications. These concepts usually cover two broad areas: the first one is the procedure or the algorithm used for requirements, specifications, implementation, verification, validation or test; that is why we usually find specific software packages used to develop specific applications which share common features as for database or web-based applications.

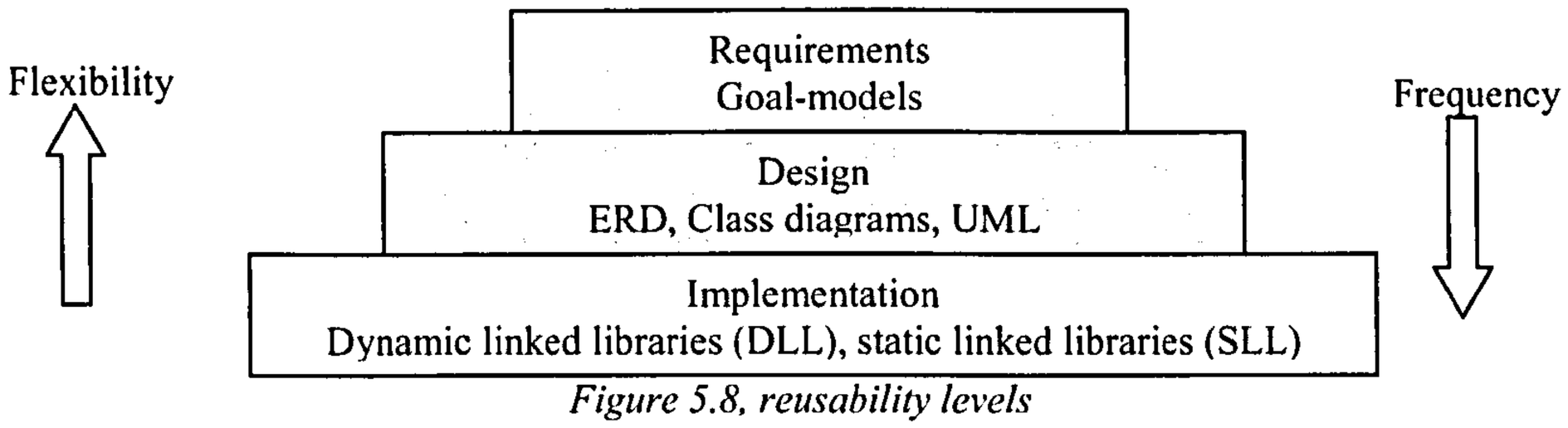
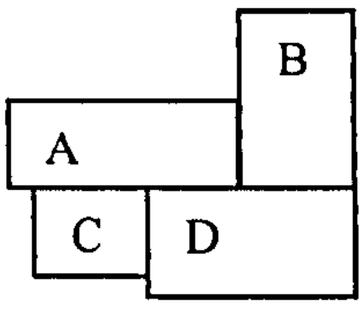


Figure 5.8, reusability levels

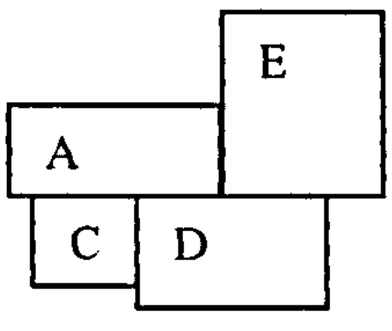
The second area is the data or the pieces of information, which constitute the requirements, specifications or the implementations. Having created GOPCSD checks and tests, which are tailored for the process control systems, we have covered only the reusability area of systemic development and testing. However, we need a library and importing system for reusing the information pieces. Since GOPCSD involves gathering the requirements and structuring, we should start the reusability of the applications information at the requirements level; saying that, we do not discourage reusing segments of the specifications or implementation, but encourage reusability as early as possible.

In figure 5.8, we sketch the reusability at the requirements, specifications, and implementation levels; although reusing requirements information is more flexible than for specifications and implementation, it is less usual to find information reused at this early level. To realize reusability at the requirements level, the system engineer may use some standard analyses, extend similar applications' requirements or even copy parts the requirements from similar applications.

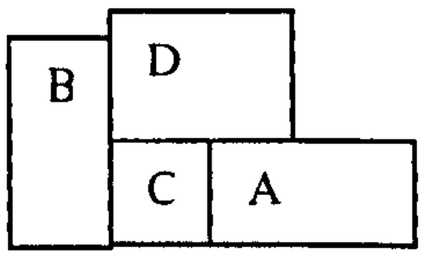
Reusability has different motivating sources within process control systems, and other systems that are built of physical components that can be rearranged differently to yield different high-level tasks.



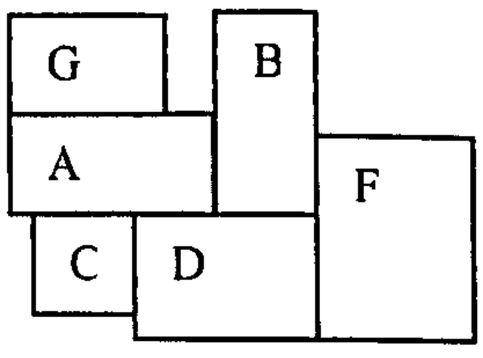
(I) An application built of an arrangement of components



(II) Replacing a component by another (E instead of B) still having the same system goals



(III) Having different arrangements for the components from the original arrangement in (I)



(IV) Adding new components (G and F) to the basic application

Figure 5.9, motivations of reusability

Figure 5.9 illustrates an application composed of four components A, B, C, and D. The specific arrangement of the components affects the overall function of the system. Another application may differ partially from the application in (I) by replacing component B by another component E, which is a slightly different one, for example, in a production cell replacing one robot model by another model. The overall application still has the same main required functions and the same constraints are to be obeyed. In (III) one can rearrange the components to construct a different application with different main functions but with the same low level functions required from the components. Finally in (IV), one can extend the application in (I) by adding new components to construct a higher-level application than (I); the new application in (IV) will contain the main functions as (I) as sub-functions, as well as the low level functions of the same components as in (I).

Motivated by these reusability concepts and following the ideas of Reubenstein and Waters in [Reubenstein and Waters 97], where they based a primary requirements classes to reuse them through inheritance, we could identified two proposed sources of reuse goal-models as follows:

- Similar applications are built of the same kind of physical components; although the components are contained in applications of different high-level goals and overall functions, the same low-level goals and basic functions are still expected from them; for example, a boiler system and a gas burner may contain similar gas valves; the overall functions derived from both applications are different, but the valve component still has the same basic requirements goals.
- Similar applications have similar high-level goals, even if they have different components. For example, a robot whose arms are devised with magnetic handling techniques has the same high-level goals as another robot with mechanical grabbers.

Figure 5.10 shows a complete goal-model with two highlighted sub-trees that can be reused in similar applications.

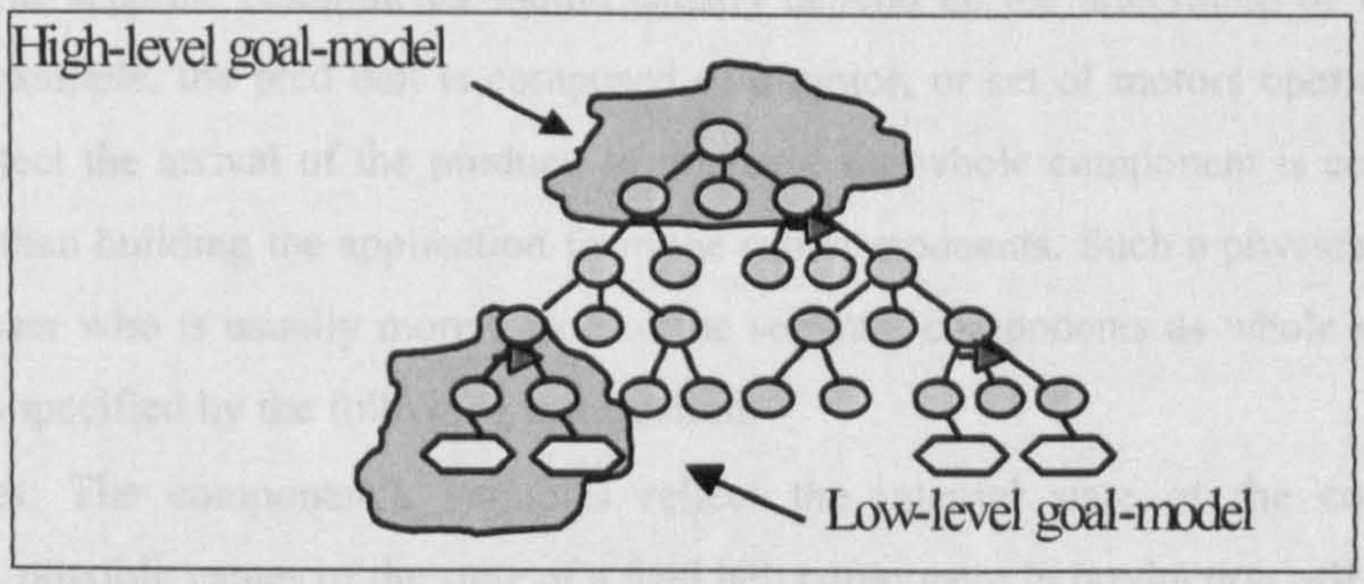


Figure 5.10, highlighting sites of reuse interest within a complete goal-model

Thus, the library can be organized as follows:

- Each library is considered as a family of related applications, like gas burners, chemical reactors, production cells.
- Each library has a list of applications, for example the production cell library includes applications like simple production cell, double-press production cell, and fault-tolerant production cell.
- Each application contains a list of high-level goal-model templates that constitute the main functions defined via such an application.

- In addition to the high-level templates, the library has a list of components that represent the building blocks of its applications, like valves and switches.
- Each physical component has low-level goal-models, variables, and agents to control such variables.

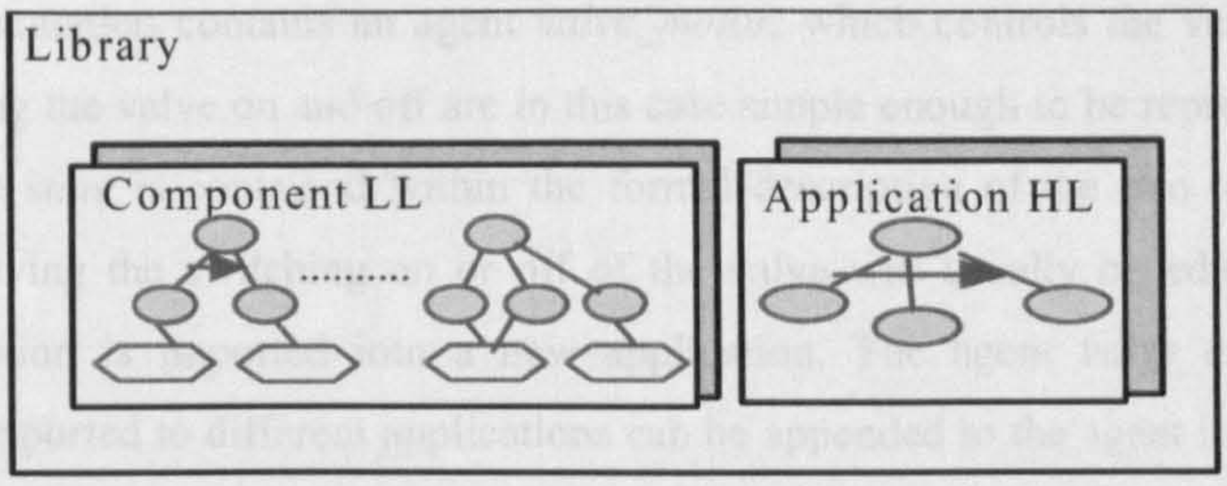


Figure 5.11, contents of one library

Figure 5.11 shows a diagrammatic sketch of a library of a specific application family; the library has a number of components and each of these has variables, agents, and goal-models; and a number of high-level templates each with an associated variables, agents (very rare to appear inside the template) and goal-models that may be incomplete but has one main goal which will be mapped into one of the application goals. This approach, which we have adopted in the GOPCSD library, is similar to product line concepts [Bosch 99, Nord Y2K, Nielmela and Ihme 01, Batory et al. 02], in the sense that it provides goal-model templates that can be refined differently for applications from the same family (e.g., different models of production cells with variant components).

5.4.1 Library Components

As indicated above, the applications belonging to a specific library share similar components. These components can be physically identified, like robots, valves, and smoke sensors. Building the application from the separate components should usually depend on the integration of the individual components; for example, the feed belt is composed of a motor, or set of motors operating together, and sensors to detect the arrival of the product; in this case the whole component is considered as a single unit rather than building the application from the sub-components. Such a physical view guides the systems engineer who is usually more aware of the separate components as whole units. Each of the components is specified by the following information:

- **Variables**. The component's variables reflect the internal state of the component, for example, possible values of the state of a feed belt component in production cell library.
- **Agents.** The component agents are either the devices, human or software modules, that are capable of controlling the component's internal state.
- **Goal-models.** The component goal-models are usually small goal-trees that describe how to realise the operation of the component. The goals within the component goal-models reference the component variables; and, the terminal goals are usually assigned to the component's agents.

When importing these components to the applications under construction, an extra process may be required to either map the imported component's variables and agents to the application's

variables and agents, respectively, or alternatively, to append the component's variables and agents to the application's lists in order to be able to reference them later from the appropriate goals.

For example, figure 5.12 represents a valve component that can be used in different applications; the valve has an on/off variable *state* to store its state. In addition to the variable, the component's representation contains an agent *valve_motor,* which controls the valve. The two goal-models for switching the valve on and off are in this case simple enough to be represented by terminal goals. The variable *state* is contained within the formal description of the two terminal goals. The condition for achieving the switching on or off of the valve will usually be edited at the time the component description is imported into a new application. The agent *valve_motor* of the valve component when imported to different applications can be appended to the agent list. Or alternatively, the agent *valve_motor* can be mapped to one of the existing application agents, *application-motor1*. In this case the *switch-on* and *switch–off* goals will be assigned to *application-motor1* agent.

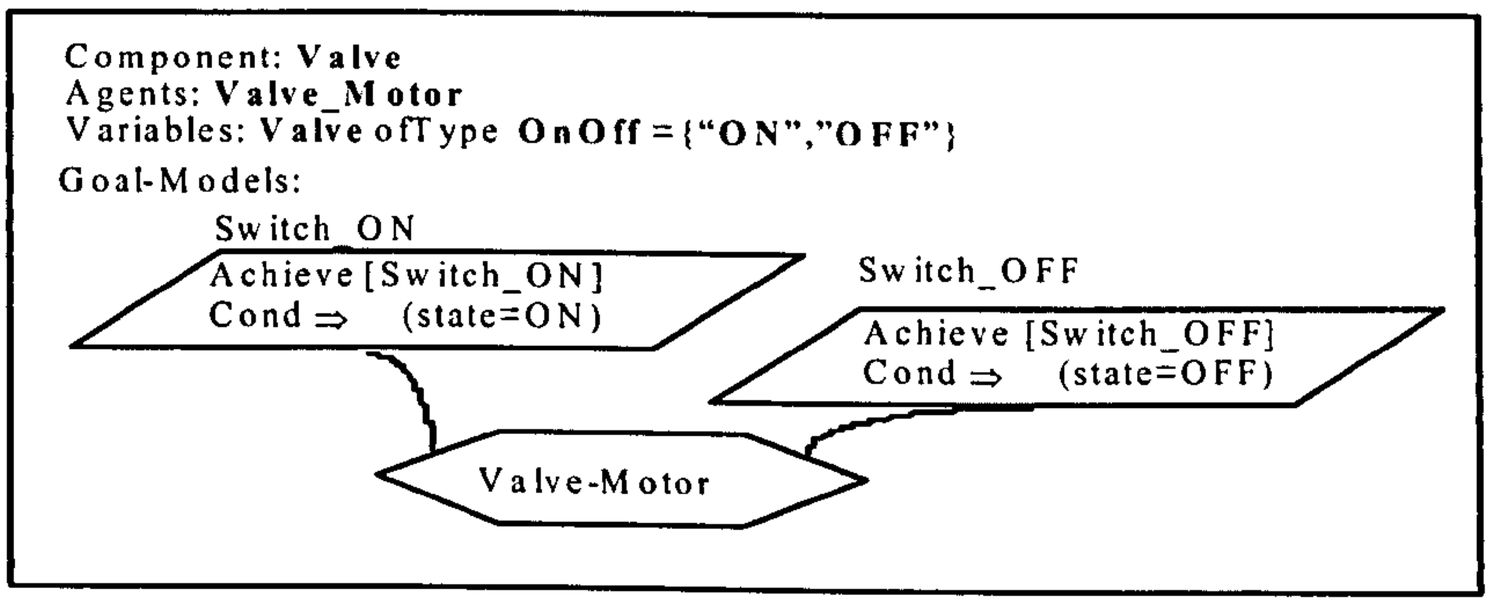


Figure 5.12, an example of a component within the library

5.4.2 Library templates

In addition to the components, each application has a number of high-level goal-model templates; these templates usually describe the high-level functions of the application without specifying the details of how to achieve these functions. Through this level of abstraction, the flexibility of using the goal-model template can be increased since it provides freedom in allowing the user to specify the low-level goals according to the current application.

For example, in a production cell, a goal like process the products or deliver the finished products can be categorized as a high-level function derived from the production cell; and hence, it should be stored in the library. Once again, these goals should not describe how to process the products, so as to increase the flexibility of using the goal with different production cells. The high-level requirements templates of the applications contain:

- **Goal-models.** The high-level goal-models templates are usually small in size and incomplete goal-trees that draw the outlines of the basic functions derived from the application.
- **Variables**. The template variables are usually application variables rather than component variables that describe the state of the application; these variables will be mapped/appended to a new application's variables during the importing process.
- **Agents.** The template agents are normally nor present, only in cases where the high-level templates have terminal-goals.

In figure 5.13, we sketch a goal-model template for a gas burner system; the goal-model describes the main functions derived from the gas burner system. This has three main functions: start up, shut down, and keep the flame burning. Only one of these three functions can be invoked by the burner system at a time; this decision depends on two conditions: whether the burner flame exists or not and whether the burner user commanded switch off or on.

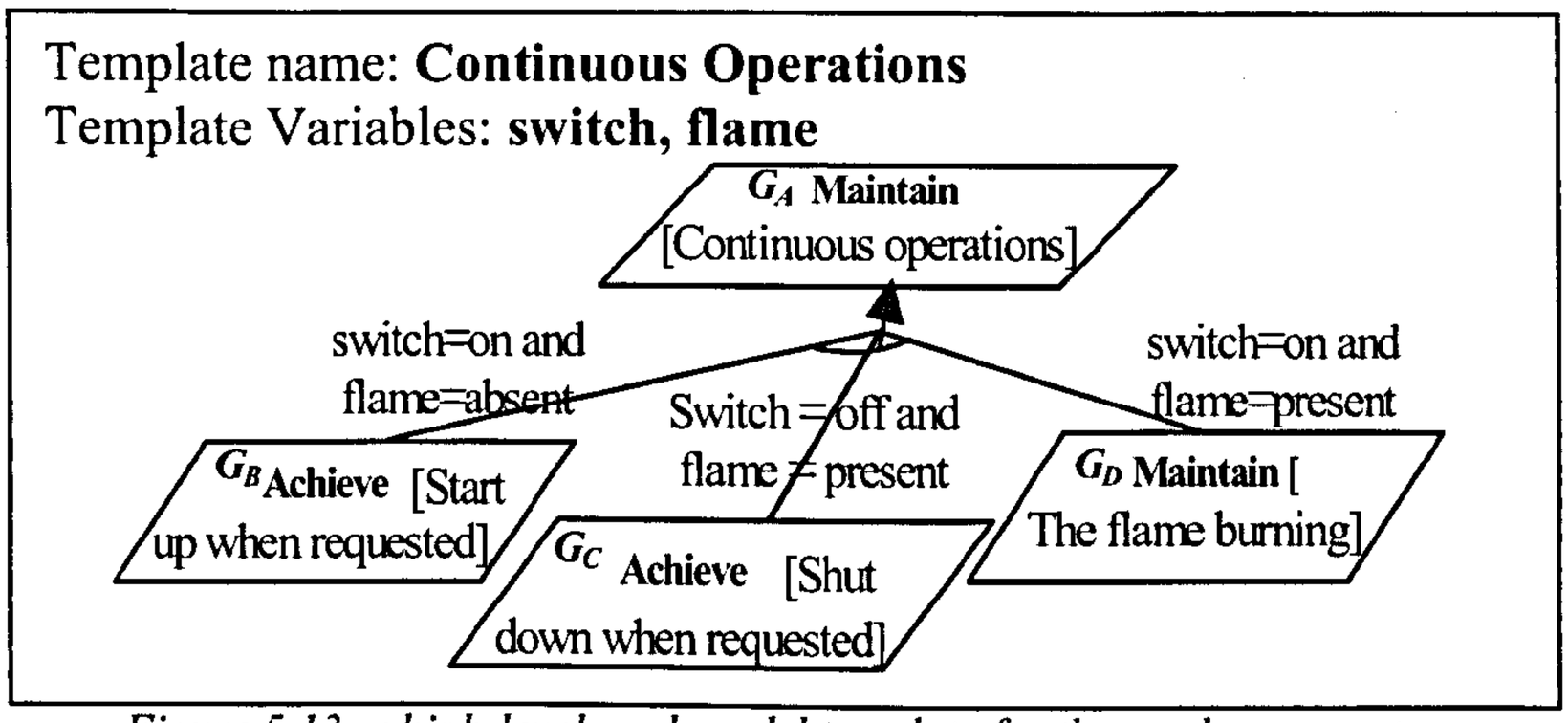


Figure 5.13, a high-level goal-model template for the gas burner system

Furthermore, to increase reusability, we suggest the library design should consider the following issues:

- The common components will be shared among the various applications to reduce the size of the library by having a single copy of the repeated component; this also will be of considerable aid in the special case when the details of the component need to be modified.
- The use of an appropriate level of abstraction in the component template would be of considerable aid for the user in order to easily build the required application; for example, a gas valve and a liquid valve can be both abstracted to a valve component that has basic open and close goals and a variable describing its state. As long as the details of the components do not affect the operations, it should be better to hide the details, especially at such an early level of requirements.

5.4.3 Hardwired Reusability Support

Hardwired reusability support is embedded within the GOPCSD tool; It is reflected on how the tool should be built; for example, choosing the goal refinement patterns, goal temporal types, agent types, obstacle classes, and integrating the appropriate validation and checks (completeness check, obstacle analysis, and goal-conflict analysis) contribute to standardize the procedure of developing process control applications. Thus, because features were captures from the procedures of developing previous process control applications, the reusability support is realised through implementing these common features within the tool.

5.5 Semantics of the goal-model

Expressing the goal-model formally can provide better understanding of the refinement, combination, and agent assignment processes. Moreover, defining formal semantics for the goal refinements will help in implementing the various algorithms of the GOPCSD tool. In this section,

most of the analysis rules assume the goal-models are complete before splitting or checking them; hence, in the following sub-sections we defined the related functions and predicates.

5.5.1 General Refinement Functions

We define general functions valid for all the refinement relationship patterns. For the purposes of these definitions, we assume that we have fixed the goal–model (i.e. the goal-model is completely refined).

- **GoalSpace** is a function that returns a set of all goals contained in the goal- model.
- **Terminal(G)** is a Boolean function that determines whether goal G is a terminal goal or not.
- **TopLevel(G)** is a Boolean function that determines whether goal G is the highest-level goal (the root of the goal-model) of the goal-model or not.
- **Ltype(G)** is a partial function mapping a goal G to the logical refinement pattern between its sub-goals. The *Ltype* function has the following range {Alternative, Sequence, Conjunction, Disjunction, Simultaneous, Inheritance}. For terminal goals, the *Ltype* is undefined since the definition is derived from the refinement relationship between the sub-goals. In fact, the domain of function *Ltype* is (*GoalSpace -{G: Terminal(G)=true}*), i.e. all non-terminal goals.
- **SubG(G)** is a function mapping a goal G to the list of its sub-goals.

$$\forall G\ [(SubG(G) = \phi) \Rightarrow (Terminal(G) = true)] \quad \textbf{5.1}$$

In 5.1, we express the property that terminal goals are defined by having no sub-goals; this rule is considered valid for complete goal-models. It can be violated before the completion of the goal-model, when some non-terminal goals need further refinement.

- **GsubG(G)** is a function mapping a goal G to the list of the descendant (direct or sub-goals of sub-goals to different levels) sub-goals of goal G.

$$\forall M[M \in GsubG(G) \Leftrightarrow M \in SubG(N) \vee \exists N\ [M \in SubG(N) \wedge N \in GsubG(G)] \quad \textbf{5.2}$$

In 5.2, it shows that if goal M is a descendant sub-goal of goal G then either it is a direct sub-goal or there is a third goal N so that M is a direct sub-goal of N and N is a descendant sub-goal of goal G.

$$\forall G\ [Terminal(G) = true \Leftrightarrow GsupG(G) = \phi] \quad \textbf{5.3}$$

In 5.3, we express the fact that terminal goals have no descendant sub-goals.

- **SupG(G)** is a function mapping each goal G to its parent goal; the range of the function is (*GoalSpace –{G: Terminal(G) =true} ∪ {null}*), i.e. non-terminal goal and *null.* The null value is required to indicate that the top-level goal has no parent goal.

$$\forall G[\forall M\ [G \in SubG(M) \Leftrightarrow M = SupG(G)]] \quad \textbf{5.4}$$

$$\forall G\ [\neg Toplevel(G) \Rightarrow G \in SubG(SupG(G))] \quad \textbf{5.5}$$

$$\forall G[\forall N[\forall M[\ N=SupG(G) \wedge M = SupG(G) \Rightarrow M = N\ ]]] \quad \textbf{5.6}$$

In 5.4, we state that if goal G is a sub-goal of goal M then M is the parent goal of G and vice versa. In 5.5, we express the fact the composition of the two functions *SubG*, *SupG* on a goal G returns the set of

goals consisting of the sub-goals of the parent goal of G including G itself. In 5.6, we define the super goal to be unique for each goal.

$$\forall G\ [TopLevel(G) = true \Leftrightarrow SupG(G) = null] \quad \textbf{5.7}$$

In 5.7, we express the fact that top-level goal has no parent goal; again this assumes the goal-model is complete and the user does not have to combine these goals into a higher-level goal.

- **GsupG(G)** is a function mapping a goal G to the set of all the ancestor goals of goal G.

$$\forall M[M \in GsupG(G) \Leftrightarrow M = SupG(G) \vee \exists N\ [M = SupG(N) \wedge N \in GsupG(G)]\ ] \quad \textbf{5.8}$$

In 5.8, we state that if goal M is an ancestor of goal G then either it is a direct parent or there is a third goal N, where M is a the parent of goal N and N is an ancestor of goal G.

$$\forall G[[TopLevel(G) = TRUE \Leftrightarrow GsupG(G) = \phi] \quad \textbf{5.9}$$

In 5.9, we express the fact that the top-level goal has no ancestors.

- **PdrG(G)** is a function that returns the predecessor goal of a given goal G provided that the parent goal is refined through Sequence pattern. If goal G is the first child, or the parent goal of G has pattern other than sequence, the *PdrG* function returns null.
- **GM(G)** is a function that returns the goal-model containing the goal G.

5.5.2 Agent assignment and control functions

In this section, we define functions related to agent assignment and variable controllability.

- **Agent(G)** is a partial function mapping a terminal goal G to an agent A that accomplishes the goal G and controls the output variables via the action formula of goal G; the range of the function Agent is the set of all agents related to the application.

$$\forall G\ [Terminal(G) = true \Leftrightarrow Agent(G) \neq null] \quad \textbf{5.10}$$

In 5.8, the fact that terminal goals must have an agent to accomplish them is represented using a restriction on the agent function value. Moreover, the rule restricts the value returned by function *Agent* to be null for the non-terminal goals.

- **Controls(A,V)** is a predicate that denotes that agent A controls variable V.

$$\forall V[\ \forall A1\ [\forall A2\ [Controls(\ A1, V) \wedge Controls(\ A2, V) \Rightarrow (A1 = A2)\ ]\ ]\ ] \quad \textbf{5.11}$$

In 5.9, the uniqueness of control concept is represented through forcing the number of agents that controls one variable in the same goal-model to be only one.

- **Contains(G,V)** is a predicate that means goal G contains variable V in its action part.

$$\forall V[\forall G[\ Terminal(G)=true \wedge Contains(G,V) \Rightarrow Controls(Agent(G), V)]\ ] \quad \textbf{5.12}$$

In 5.12, we relate the output variable V contained in a terminal goal G to the Agent A that accomplishes the goal; any output variable contained in the goal action formulae will be controlled by the agent A.

5.5.3 Refinement Relationships Properties

In this section, we define some properties for the specific refinement patterns. Laws 5.13 to 5.31 are a list of laws governing the refinement relationships between the different goals involved. The

laws are grouped into four categories depending on their nature: general definition, Commutativity, Associativity, and Distributivity.

5.5.3.1 General Refinement definition

Here we define the different refinement patterns we are going to use in the following sections. In addition, we relate the general functions in 5.5.1 to the refinement patterns.

$\forall G\ [G=\text{Alternative}\ (g_1,..,\ g_n) \Rightarrow G = \text{SupG}(g_1) \wedge .. \ G = \text{SupG}(g_n) \wedge g_1 \in \text{SubG}(G) \wedge .. \ g_n \in \text{SubG}(G) \wedge \text{Ltype}(G) = \text{Alternative}]$ **5.13**

$\forall G\ [G=\text{Sequence}\ (g_1,..,\ g_n) \Rightarrow G=\text{SupG}(g_1) \wedge .. \ G=\text{SupG}(g_n) \wedge g_1 \in \text{SubG}(G) \wedge .. \ g_n \in \text{SubG}(G) \wedge \text{Ltype}(G) = \text{Sequence} \wedge \text{PrdG}(g_1) = \text{null} \wedge .. \wedge \text{PrdG}(g_n) = g_{n-1}\ ]$ **5.14**

$\forall G\ [G=\text{Conjunction}\ (g_1,..,\ g_n) \Rightarrow G = \text{SupG}(g_1) \wedge .. \ G=\text{SupG}(g_n) \wedge g_1 \in \text{SubG}(G) \wedge .. \ g_n \in \text{SubG}(G) \wedge \text{Ltype}(G) = \text{Conjunction}\ ]$ **5.15**

$\forall G\ [G=\text{Disjunction}\ (g_1,..,\ g_n) \Rightarrow G = \text{SupG}(g_1) \wedge .. \ G=\text{SupG}(g_n) \wedge g_1 \in \text{SubG}(G) \wedge .. \ g_n \in \text{SubG}(G) \wedge \text{Ltype}(G) = \text{Disjunction}\ ]$ **5.16**

$\forall G\ [G=\text{Simultaneous}\ (g_1,..,\ g_n) \Rightarrow G = \text{SupG}(g_1) \wedge .. \ G=\text{SupG}(g_n) \wedge g_1 \in \text{SubG}(G) \wedge .. \ g_n \in \text{SubG}(G) \wedge \text{Ltype}(G) = \text{Simultaneous}]$ **5.17**

$\forall G\ [G=\text{Inheritance}\ (g) \Rightarrow G=\text{SupG}(g) \wedge \text{SubG}(G)=\{g\} \wedge \text{Ltype}(G) = \text{Inheritance}\ ]$ **5.18**

Rules 5.13, 5.14, 5.15, 5.16 and 5.17 define the six refinement patterns. They relate the definition alternative, sequence, conjunction, disjunction, simultaneous, and inheritance, respectively to *SupG*, *SubG*, *PdrG* and *Ltype* functions.

5.5.3.2 Commutativity Properties

This section lists the refinement relationships' commutativity properties as follows:

$\forall G1\ [\forall G2[\text{Alternative}\ (G1, G2) \equiv \text{Alternative}\ (G2, G1)]]$ **5.19**

$\forall G1\ [\forall G2[\text{Conjunction}\ (G1, G2) \equiv \text{Conjunction}\ (G2, G1)]]$ **5.20**

$\forall G1\ [\forall G2[\text{Disjunction}\ (G1, G2) \equiv \text{Disjunction}\ (G2, G1)]]$ **5.21**

$\forall G1\ [\forall G2[\text{Simultaneous}(G1, G2) \equiv \text{Simultaneous}(G2, G1)]]$ **5.22**

$\neg\forall G1\ [\forall G2[\text{Sequence}\ (G1, G2) \equiv \text{Sequence}\ (G2, G1)]]$ **5.23**

Laws 5.19, 5.20, 5.21 and 5.22 can be used to have different refinement forms of the same goal-model, while law 5.23 means that for the Sequence refinement pattern, the order of the sub-goals must be considered.

5.5.3.3 Associativity Properties

This section lists the refinement relationships' association properties as follows:

$\forall G1\ [\forall G2[\forall G3[\ \text{Conjunction}\ (\text{Conjunction}\ (G1, G2), G3) \equiv \text{Conjunction}\ (G1, \text{Conjunction}\ (G2, G3)) = \text{Conjunction}\ (G1, G2, G3)\ ]]]$ **5.24**

$\forall G1\ [\forall G2[\forall G3[\text{Alternative}\ (\text{Alternative}\ (G1, G2), G3) \equiv \text{Alternative}\ (G1, \text{Alternative}\ (G2, G3)) \equiv \text{Alternative}\ (G1, G2, G3)]]]$ **5.25**

$\forall G1\ [\forall G2[\forall G3[\ \text{Disjunction}\ (\text{Disjunction}\ (G1, G2), G3) \equiv \text{Disjunction}\ (G1, \text{Disjunction}\ (G2,$

G3)) ≡ Disjunction (G1, G2, G3)]]]..... **5.26**

∀G1 [∀G2[∀G3[Sequence (Sequence (G1, G2), G3) ≡ Sequence (G1, Sequence (G2, G3)) ≡ Sequence (G1, G2, G3)]]]..... **5.27**

Laws 5.24, 5.25, 5.26, and 5.27 can be used to optimise the goal-model through reducing the number of the nested levels; for example, if goal G1 is defined as Sequence(G2, G3), and goal G3 as Sequence(G4,G5), then goal G1 will be equivalent to Sequence (G2, G4, G5).

5.5.3.4 Distributivity Properties

This section lists the refinement relationships' distribution property as follows:

∀G1 [∀G2[∀G3[Sequence (Alternative (G1, G2), G3) ≡ Alternative (Sequence (G1, G3), Sequence (G2, G3))]]]..... **5.28**

∀G1 [∀G2[∀G3[Conjunction (Alternative (G1, G2), G3) ≡Alternative (Conjunction (G1, G3), Conjunction (G2, G3))]]]..... **5.29**

∀G1 [∀G2 [∀G3 [Simultaneous (Alternative (G1, G2), G3) ≡ Alternative (Simultaneous (G1, G3), Simultaneous (G2, G3))]]]..... **5.30**

∀G1[∀G2[∀G3[Disjunction (Alternative (G1, G2), G3) ≡ Alternative (Disjunction (G1, G3), Disjunction (G2, G3))]]]..... **5.31**

Laws 5.28, 5.29, 5.30 and 5.31 explain how to move the alternative refinement pattern upwards in the goal-model tree.

5.5.4 Special Refinement Pattern Restrictions

In this section we express some restrictions we place on two refinement patterns.

5.5.4.1 Inheritance Refinement Pattern Restriction

As early mentioned in section 5.3.6, we based a restriction on the inheritance refinement pattern as follows:

∀G [G = Inheritance (G1) ∧ G = Inheritance (G2)⇒((GM(G1) = GM(G2)) ∧(G1=G2)) ∨ (GM(G1) ≠GM(G2))]..... **5.32**

Law 5.32 is a restriction for the inheritance refinement; it states that if two sub-goals are inheritance refinements of the same super goal, either they belong to different goal-models or they are the same; in other words, it restricts the number of sub-goals whose refinement relationship is inheritance to only one per parent goal in the same goal-model.

5.5.4.2 Simultaneous Refinement Pattern Restriction

The simultaneous refinement pattern is restricted using the following rule as follows:

∀G [G = Simultaneous(G1, .. Gn) ⇒ Terminal(G1) = true ∧ .. ∧ Terminal(Gn) = true]..... **5.33**

In law 5.33, we restrict the simultaneous refinement by forbidding the sub-goals to be non-terminals. This restriction can give the user more control in firing the goals simultaneously; otherwise if one of the goals needs refinement then the conjunction pattern can be more effectively used.

5.5.5 Splitting compound goal-models

This section defines the rules that can be used to split the compound goal-model into equivalent number of simple goal-models

Split ≡ Goal → P(Goal)	**5.34**
P(goal) means a power set whose elements are of type goal	
Fn (G1 × G2) ≡ {Fn (x_1, y_1) , Fn (x_2, y_1) , … ,Fn (x_1, y_2) , … Fn (x_n, y_m) }	**5.35**
Where Split(G1) = { x_1, … x_n}, Split(G2) = { y_1, … y_m} and Fn ∈ {Simultaneous, Sequence, Disjunction, Conjunction}	
∀G1[∀G2[Split (Alternative (G1, G2)) ≡ {Split(Inheritance(G1)), Split(Inheritance(G2))}]]..	**5.36**
∀G1[∀G2[Split (Conjunction(G1, G2)) ≡ Conjunction (Split(G1) × Split(G2))]].....	**5.37**
∀G1[∀G2[Split (Disjunction(G1, G2)) ≡ Disjunction (Split(G1) × Split(G2))]].....	**5.38**
∀G1[∀G2[Split (Sequence(G1, G2)) ≡ Sequence (Split(G1) × Split(G2))]].....	**5.39**
∀G1[∀G2[Split (Simultaneous(G1, G2)) ≡ Simultaneous(Split(G1) × Split(G2))]].....	**5.40**
∀G[Split (Inheritance(G)) ≡ Inheritance (Split(G))].....	**5.41**
∀G[Terminal(G) = true ⇒ Split(G) = {G}].....	**5.42**

In 5.34, we define *Split* as a function that has goals as domain and the range of sets of goals; the *Split* function is used to derive the different solution versions for a given compound goal/ goal-model. In 5.35, we define cross product of goals under one of the refinement relationships; this is used in laws 5.36, 5.37 and 5.38. Laws 5.36, 5.37, 5.38, 5.39, 5.40, 5.42 and 5.42, can be used to implement an algorithm to split the compound goal-models.

For example, consider the goal-model of figure 5.14, where G1 = Sequence (G2, G3) and G2 = Disjunction (G4, G5), then the following equations are true:

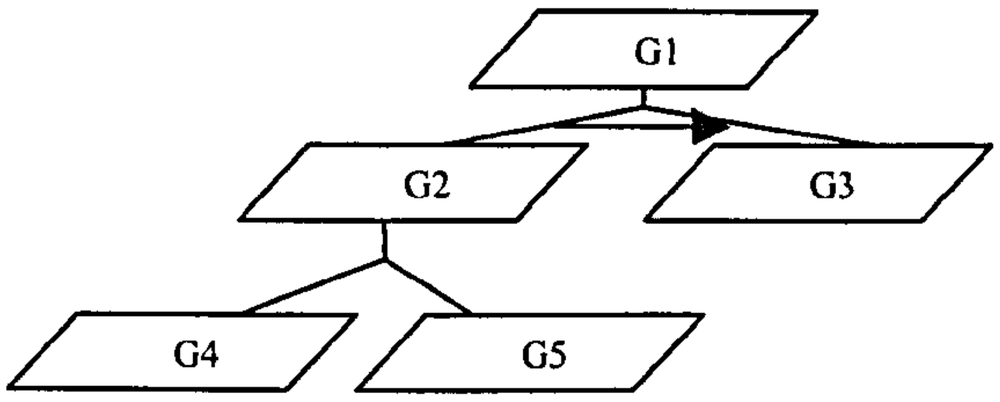


Figure 5.14, an example of a goal-model to indicate the semantics

GoalSpace ={G1, G2, G3, G4, G5}
GoalSpace – {g: Terminal(g)=true} ={G1, G2, G3}
G2 = Alternative (G4, G5) = Alternative (G5, G4)
Terminal(G3) = true, Terminal(G4) = true, Terminal(G2) = false
Toplevel(G1) = true, Toplevel(G2) = false
SubG(G1) = {G2, G3}, GsubG(G1) = {G2, G3, G4, G5}
SupG(G1) = null, SupG (G4} = G2, GsupG(G5) = {G1, G2}

PdrG(G1)= null, PdrG(G2)= null, PdrG(G3)=G2

Ltype(G1) = Sequence, Ltype(G2) = Alternative, Ltype(G3) is undefined

G1 = Sequence (Alternative (G4, G5), G3) = Alternative (Sequence (G4, G3), Sequence (G5, G3))

Split(G3)={G3} , Split(G2)= {Split(G4), Split(G5)}, Split(G4)={G4} , Split(G5)={G5}

Split(G1)= Sequence(Split(G2) × Split(G3)) ={Sequence(G4, G3), Sequence(G5,G3))}

5.5.6 Propagating the pre- and post- conditions within goal-model

The locality of goals within the goal-model usually reduce the effort required to build goal-model; in addition, it enables the user to focus on parts of the applications layer by layer, mode by mode, or situation by situation. However, to translate the goal-model into specifications, to enable conflict and completeness checking, and to animate the goal-model, one usually needs to accumulate the higher-goals' conditions to define the actual condition for a given goal. We formulate the rules for accumulating the goal conditions as follows:

- **postCond(G)** is a function that returns a Boolean expression denotes the post-condition of goal G.
- **preCond(G)** is a function that returns a Boolean expression stands for the local pre-condition of goal G.
- **accPreCond(G)** is a function that returns a Boolean expression stands for the accumulated condition of goal G due to the pre-conditions of the ancestor goals.
- **accPostCond(G)** is a function that returns a Boolean expression stands for the contribution of the predecessor goals accumulated post-conditions in the given goal G pre-condition.
- **accCond(G)** is a function that returns a Boolean expression stands for the actual pre-condition for the goal G considering the effects of the other all other predecessor and ancestor goals.

$$accPreCond(G) = \wedge_{g \in G \sup G(G)} postCond(g) \wedge preCond(g) \quad \text{5.43}$$

$$accPostCond(G) = postCond(g), \text{ if g=PdrG(G)} \quad \text{5.44}$$

$$accCond(G) = preCond(G) \wedge accPreCond(G) \wedge accPostCond(G) \quad \text{5.45}$$

$$accPostCond(G) = true, \text{ if PdrG(G) = null} \quad \text{5.46}$$

$$accPreCond(G) = true, \text{ if TopLevel(G) = true} \quad \text{5.47}$$

$$\forall G[\neg accPreCond(G) = true \Rightarrow \forall G1[G1 \in GsupG(G) \Rightarrow accPreCond(G) = true]] \quad \text{5.48}$$

Laws 5.46 and 5.47 are considered as the stopping condition for the accumulation algorithms. Law 5.43 that considers the propagation of pre-conditions stops at the top of the goal-model; while law 5.44 that consider the propagation of post-conditions due to the Sequence pattern stops at the first child of sequence refinement pattern or directly at any goal that does not refine its parent goal through Sequence pattern. Finally, law 5.45 adds the local conditions together to formulate the global condition for the goal G that will be used in checking conflict and completeness, and in animations and in the translation to B machines. Law 5.48 means that is the global pre-condition of one goal is true then each of its ancestor goals has a true pre-condition. This explains why active goals are always found as tree-branches (paths) starting from the goal-model tree root. And ending either at a terminal goal to activate its action or at a non-terminal goal that has no active sub-goal at the current circumstances.

5.5.7 Detecting goal-conflicts

To check the consistency of the goal-model, the individual goals can be checked to ensure they do not prescribe inconsistent behaviour among themselves.

- **Conflict(G1,G2)** is a predicate denotes that goals G1 and G2 conflict each other; usually, the two goals try to access the same output variable.

$$\forall G1[\forall G2[\ Conflict(G1,\ G2) \Rightarrow \exists V\ [Contains(G1,\ V) \wedge Contains(G2,V)] \wedge G1 \notin GsubG(G2) \wedge G2 \notin GsubG(G1) \wedge \neg (\ accPreCond(G1) = true \Rightarrow accPreCond(G2) = false)\ ]\ ] \quad \textbf{5.49}$$

In 5.49, it shows that if two Goals G1 and G2 conflict each other, then, there is some output variable which is contained in their action formulae, their accumulated pre-conditions are not disjoint and non of them is a descendant goal of the other. Rule 5.49 can be used to discover goal-conflict by checking the right hand side of the first imply operator.

5.5.8 Detecting unreachable goals

The reachability of a goal (goal-reachability) means the goal itself can be fired in some scenario. However, in large applications, users are more likely to err and impose logically erroneous pre-conditions of some goals, resulting in unreachable goals.

- **Reachable(G)** is a predicate that denotes goal G is reachable; or in other words under some pre-conditions the goal G will be activated.

$$\forall G[\neg Reachable(G) \Rightarrow \boxtimes[accPreCond(G) = false)]] \quad \textbf{5.50}$$

$$\forall G[\neg Reachable(G) \Rightarrow \forall G1[G1 \in GsubG(G) \Rightarrow \neg Reachable(G1)]]. \quad \textbf{5.51}$$

$$\forall G\ [\exists G1[Reachable(G1) \wedge G1 \in GsubG(G)] \Rightarrow Reachable(G)] \quad \textbf{5.52}$$

In 5.50, it shows that Gaol G will be unreachable if its pre-condition is always false. Usually, this problem occurs because the pre-condition of the unreachable goal contradicts ???????????? itself, or an ancestor goal's pre-condition or a predecessor goal's post-condition. In 5.51 and 5.52, we relate the reachability of parent and child goals; if the parent goal is not reachable the child or grandchild goal as well will not be reachable. But, if a parent goal has a reachable child or grandchild goals then the parent goal is reachable. Rules 5.51 and 5.52 can be used to reduce the time required for the detection of unreachable goals. Furthermore, the rules can be used to determine the main source of unreachability.

5.5.9 Checking the completeness of goal-models

The completeness of a goal-model over a set of variables means that every possible combination of these variables is considered. In particular, regarding the nature of a controller, for each possible input variable combination, each output variable should be fully determined. In other words, given any possible variable combination, each output variable value should be defined.

- **Complete(G)** is a predicate that denotes goal G is complete.

$$Complete(G) \Rightarrow \Box[\forall V[\ \exists g\ [\ g \in GsubG(G) \wedge Terminal(g) = true \wedge accPreCond(g) = true \wedge Contains(\ g, V)\ ]\ ]\ ] \quad \textbf{5.53}$$

In 5.54, we define completeness of goal G as following: the goal G is considered complete if for each variable combinations there exist a goal *g* is a terminal goal, a descendant sub-goal of goal G, its accumulated pre-condition is fired and the variable V is controlled within the goal *g* action formulae. Furthermore, the variable values should not be dependant on its initial value.

5.6 Conclusions

The GOPCSD method covers the early requirements stages that include: creating abstract requirements, refining and formalising the requirements and checking, validating, debugging, and translating them into formal specifications. Some adaptations have been introduced to the KAOS method to fit process control applications. We have identified six recurring goal-refinement patterns and recommended when to use each of them. The GOPCSD method approach to reusability support and the construction of the requirements library have been discussed. Finally, we established the formal semantics for the goal-model in order to use it in implementing the various algorithms of the GOPCSD tool.

Detailed Features of the GOPCSD tool

6

In this chapter, we describe the details of the GOPCSD tool. Furthermore, we describe how the GOPCSD tool can guide the user in constructing, checking and translating the requirements using the different features and checks offered by the tool.

6.1 Introduction

There are a number of tools based on the KAOS method [KAOS, Objectiver/Grail], which adopt the concepts of goal-driven methods and combining the informal and formal description of the application and its local environment. However, there are a number of reasons that made us choose to develop our own formal model and corresponding testing and checking algorithms. First, because the GOPCSD method attempts to build an independent requirements stage, the formal model (which is based on pre- and post- condition of the goals and the refinement patterns of the non-terminal goals) should be directly related to the goals the user created/modified himself/herself; this can give a better chance to correct the requirements model rather than use the requirements model as an intermediate representation, which entails difficulties in tracing the errors and correction backwards to and forwards from the requirements. In addition, in the domain of process control systems, the systems engineer can have better awareness of control systems and the situations that can occur during runtime, because the system is usually built by integrating existing components. This encourages us to integrate completeness and reachability checks that can exhaustively test the control systems. However, we

could not use the underlying formal model of the KAOS method because it is too general (including existence, for all and temporal operators, even within the post-conditions of the goals). Another important point to consider is the component-based nature of process control systems. This encourages us, as noted in chapter 5, to combine the top-down with bottom-up development. Thus, We were motivated to develop the GOPCSD tool [El-Maddah and Maibaum 03b] to support the GOPCSD method and automate the various phases of the process control requirements development. The GOPCSD tool offers an interactive and integrated environment to build and check the process control requirements, and finally translate them into a B specification. The requirements of the applications are represented as lists of the components, variables, agents, and goal-models, which are briefly described in the following sections:

6.1.1 The Components

Components represent the physical parts of the applications, such as valves, robots, and deposit belts. The detailed specifications of each component, including its variables, agents and goal-models, are stored in the GOPCSD library. The systems engineer can create/edit the component details using the GOPCSD library manager, but not within the GOPCSD tool IDE in order to ensure consistency of components across applications.

6.1.2 The Variables

Variables are considered as an essential aspect to formalise the user requirements. In the GOPCSD tool, the application global state is normally described by a set of variables. Each of these variables has one of three types: input, output or intermediate. In the GOPCSD tool, the variables are associated with the high-level goal-model templates or the components, which the user can import from the library; however, the tool's user can still create, edit, and delete variables from the application space.

6.1.3 The Agents

Agents are the objects that control the application and its local environment. Some of the agents can be part of the application to be built, like software interface programs for hardware parts, or, alternatively, they can be existing programs or hardware devices that will be responsible for accomplishing pre-defined tasks (goals) to fulfil the overall application operation. Agents can have one of the following three types: device, software and human. The main source of agents is when the user imports components from the library. But, the user can define agents (directly belonging to the application) within the GOPCSD tool development environment.

6.1.4 The Goal-models

Goal-models are the main constituent of the structured requirements; they represent the user requirements in a hierarchy of building blocks. Each goal-model starts with a main goal that has general scope to specify the overall application requirements; this goal is usually refined to a number of goals describing sub-parts, different aspects, or operation-modes of the application. In the GOPCSD tool, goal-models can be used as workspaces until the user manages to construct one complete goal-

model that specifies the whole application requirements and each of the terminal goals, of this goal-model, is assigned to an agent. Goal-models can be checked as soon as they are created in order to provide the user with feedback to establish the correct requirements as early as possible. After these repeated feedback processes, the goal-model can be animated to catch the logical bugs that have not been discovered yet. Finally, the user can generate the formal specifications of the application by automatically translating the goal-models into B machines.

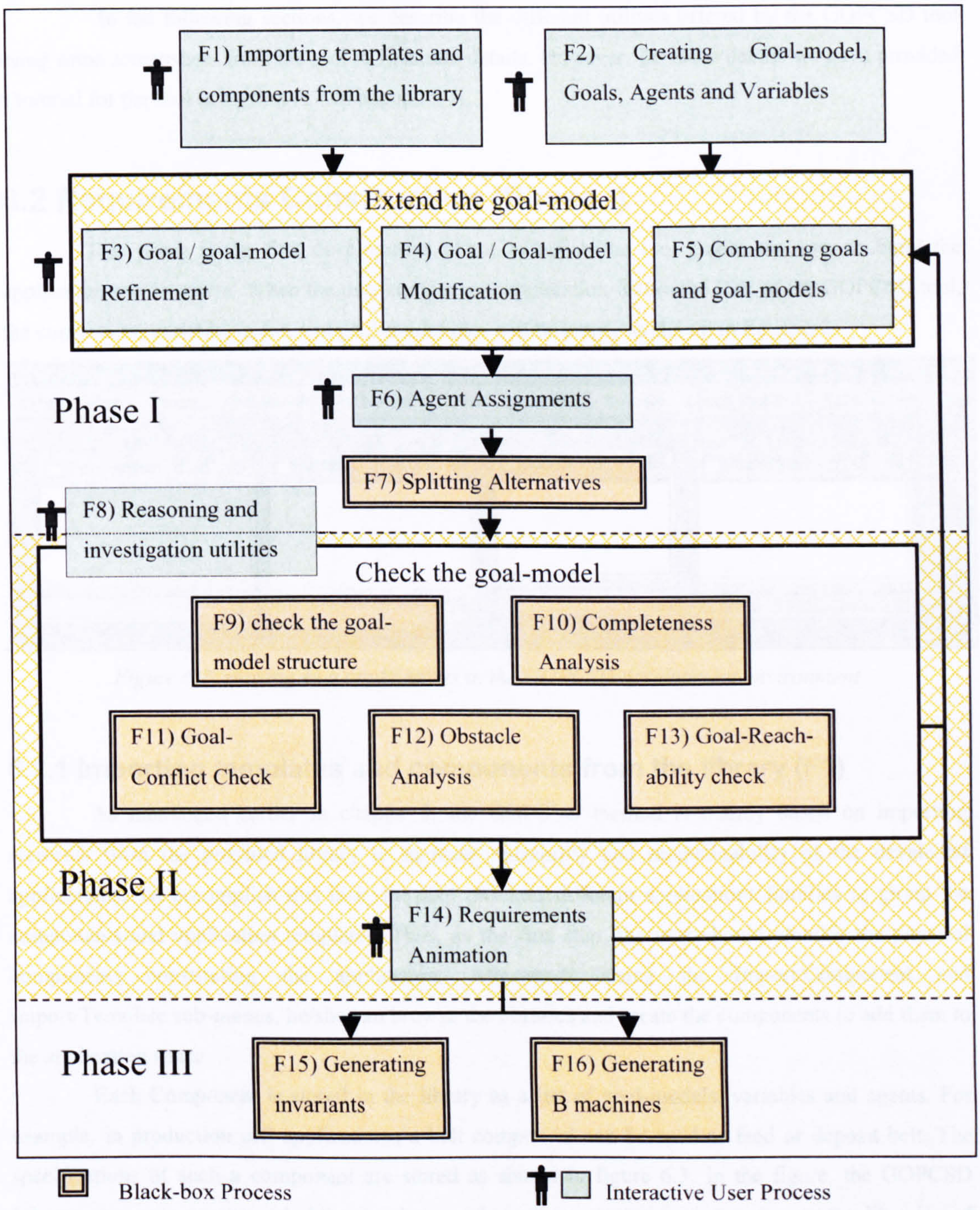


Figure 6.1, the detailed-function decomposition of the GOPCSD tool

The GOPCSD tool can be regarded as an integrated development environment (IDE) for process control requirements development. This IDE offers various functions like importing, editing, examining, reasoning about parts of, animating, and compiling goal-models. These functions constitute

the three development phases described in figure 5.1. Having briefly described the requirements development lifecycle in the GOPCSD method in chapter 5, we continue describing the GOPCSD tool in detail in this chapter. Figure 6.1 is a schematic diagram of the GOPCSD tool functions, represented as boxes labelled with their brief descriptions. The arrows connecting the different functions denote the data and control flow. Some of the functions have a human icon beside them to indicate user-interaction, while the others will be seen by the user as "black boxes".

In the following sections, we describe the different utilities offered by the GOPCSD tool using some screenshots from the tool to illustrate details. However, for more details we have provided a tutorial for the tool in section A.4 of appendix A.

6.2 Requirements Construction (Phase I)

This phase is the first development phase in which the tool guides the user to build the application requirements. When the user starts a new application within the IDE of the GOPCSD tool, the component, variable, agent, and goal-model lists will be empty as shown in figure 6.2.

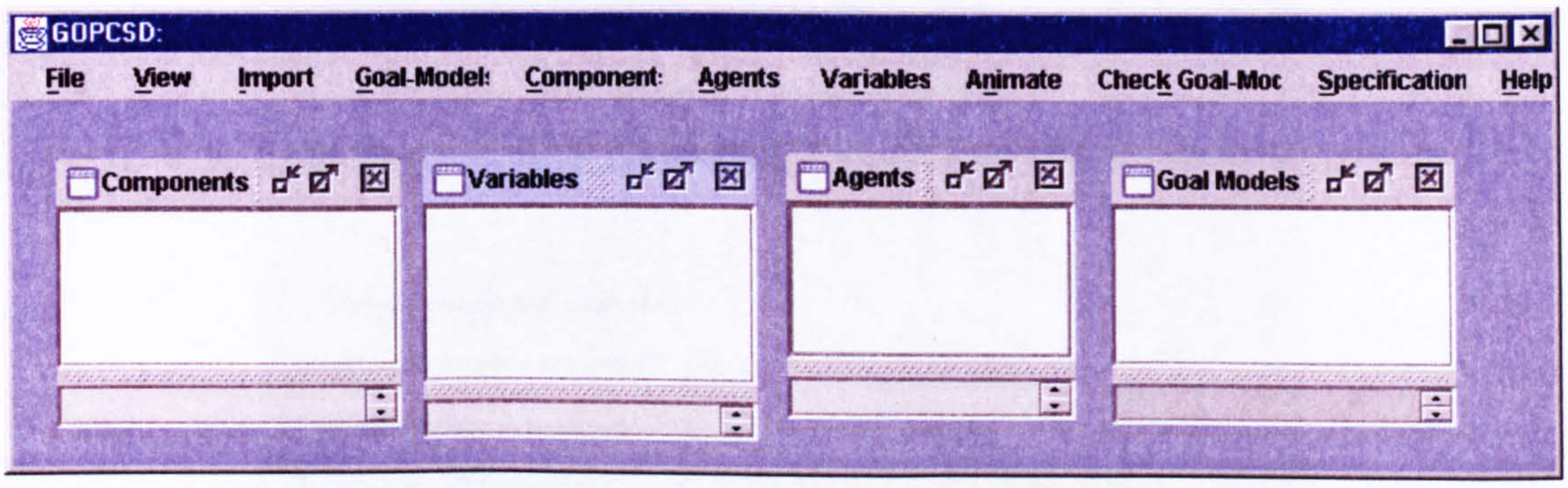


Figure 6.2, starting new applications in the GOPCSD development environment

6.2.1 Importing templates and components from the library (F1)

As mentioned earlier in chapter 5, the GOPCSD method is mainly based on importing elements from the provided library to increase reusability and maintainability of the developed applications. To accomplish this task, the tool provides different applications families to group the components and application templates. Thus, as the first step, the user should identify the physical components constituting the application. Afterwards, using the Import/Component and Import/Template sub-menus, he/she can browse the libraries and locate the components to add them to the application space.

Each Component is stored in the library as a list of goal-models, variables and agents. For example, in production cell applications, a belt component can be used as feed or deposit belt. The specifications of such a component are stored as shown in figure 6.3. In the figure, the GOPCSD Library manager program desktop is shown where the systems engineer can create libraries of components and templates in order to reuse them for different applications.

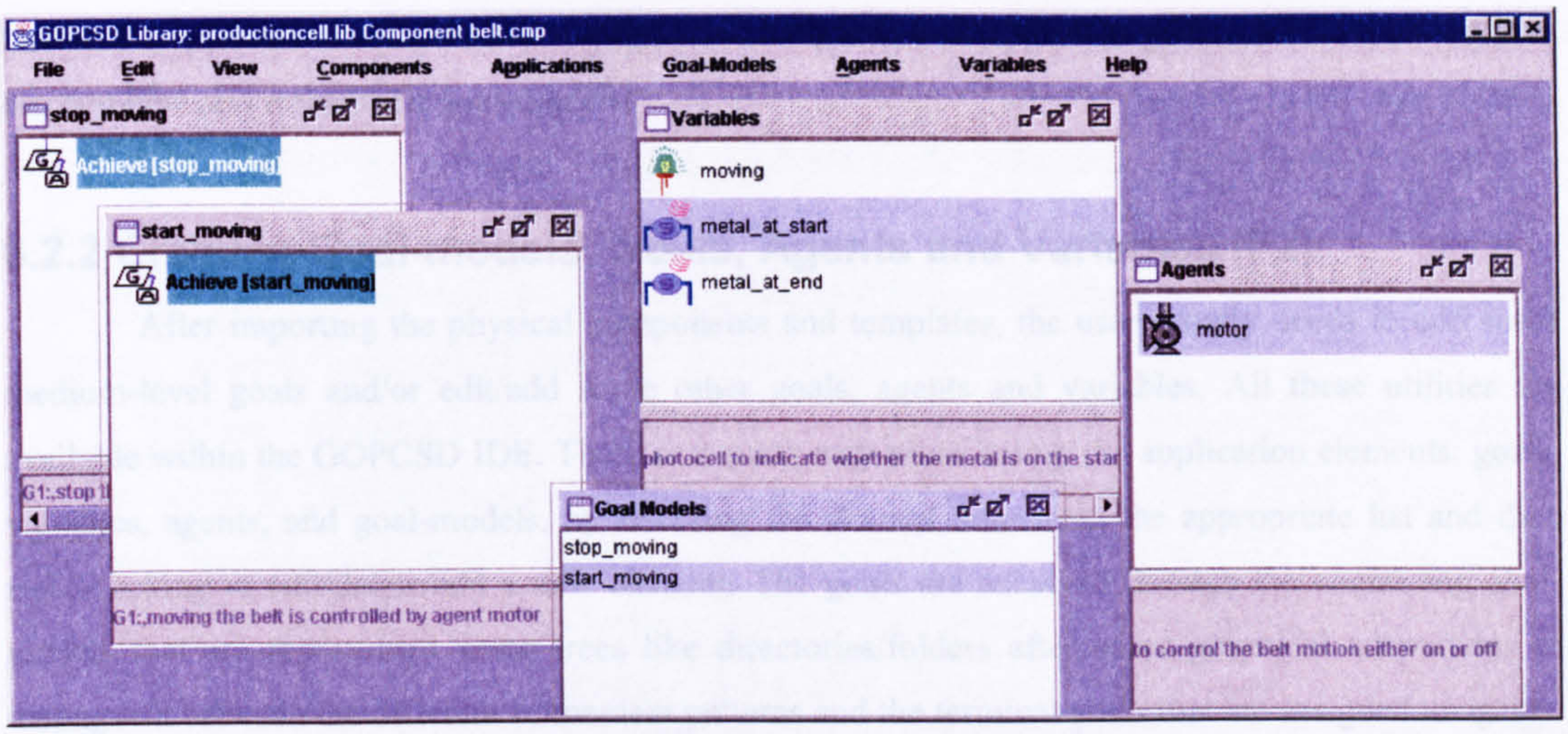


Figure 6.3, the GOPCSD library manager, a belt component.

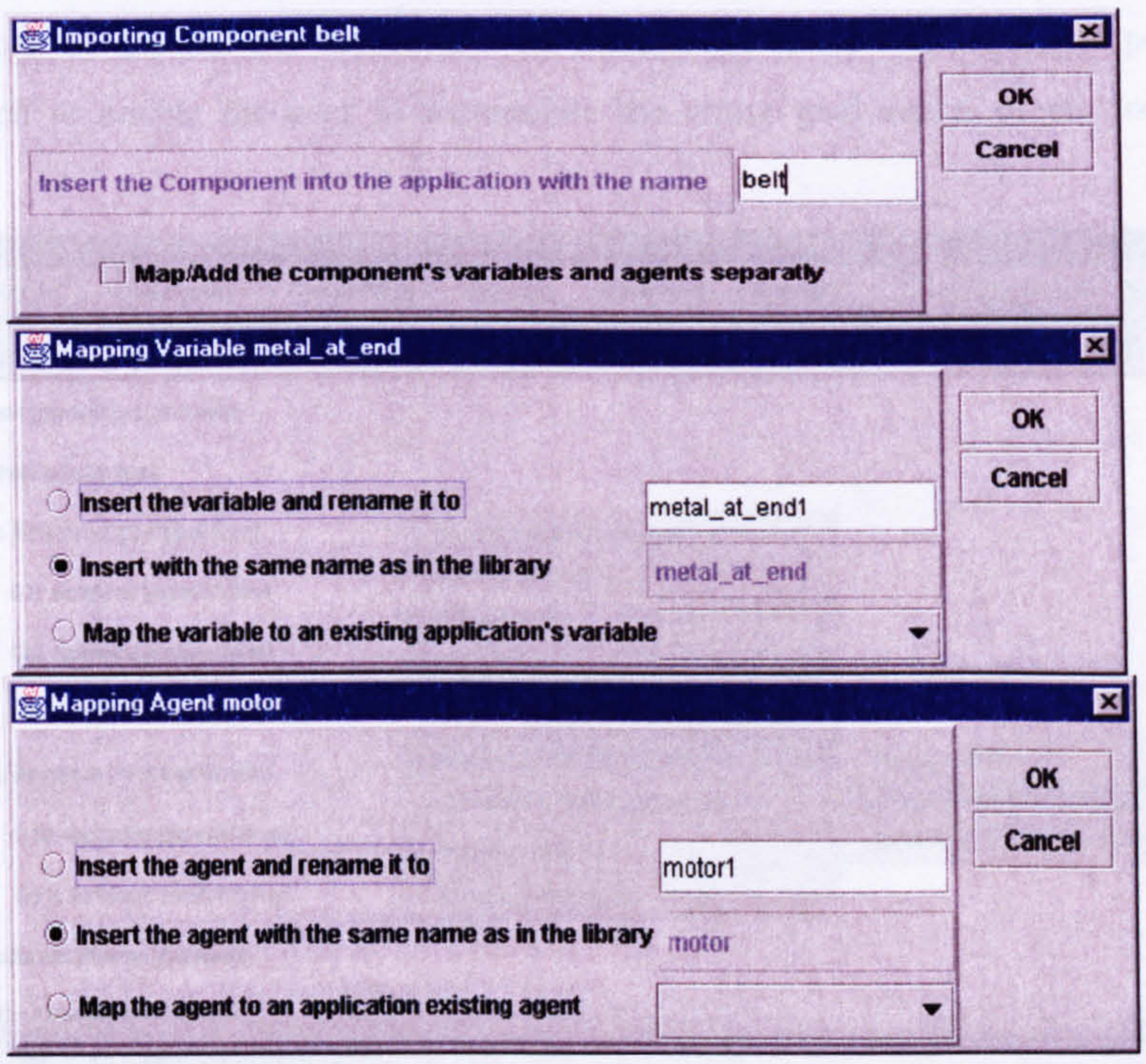


Figure 6.4, importing a component from the library and mapping/adding its details

As in figure 6.2, the GOPCSD tool enables the user to import such components using the Import/Component sub-menu. In addition, the tool allows the components' variables and agents to be added to the application space with different names or the same names that appear in the library. In some other cases, the user may need to map the library variables or the agents to some of the existing variables or agents rather than appending them to the application lists. The tool achieves such a task through employing the dialogue boxes shown in 6.4.

These dialogue boxes enable the user to rename the imported components; furthermore, the user will have a chance to "insert with new name"/insert/map each of the component' variables and agents. Similarly, the user can import the library templates and map or add their variables and agents.

Unlike the application components, the imported templates will appear as goal-models; thus, the user can combine and refine their goal-models.

6.2.2 Creating Goal-models, Goals, Agents and Variables (F2)

After importing the physical components and templates, the user usually needs to add some medium-level goals and/or edit/add some other goals, agents and variables. All these utilities are available within the GOPCSD IDE. There is a quick way of accessing the application elements: goals, variables, agents, and goal-models, by selecting the desired item from the appropriate list and then right-clicking to edit/delete/add a new element. The goals are accessed through the containing goal-models that are represented using trees like directories/folders after some graphical adaptations to distinguish between the different refinement patterns and the terminal goals that are assigned to agents. Figure 6.5 shows the desktop of the GOPCSD tool focusing on one goal-model frame, where the different goals can be noticed and identified with numbers (like G1 and G12); the pop up menu is displayed as well to enable the user to manipulate the entire goal-model or the individual goals, separately.

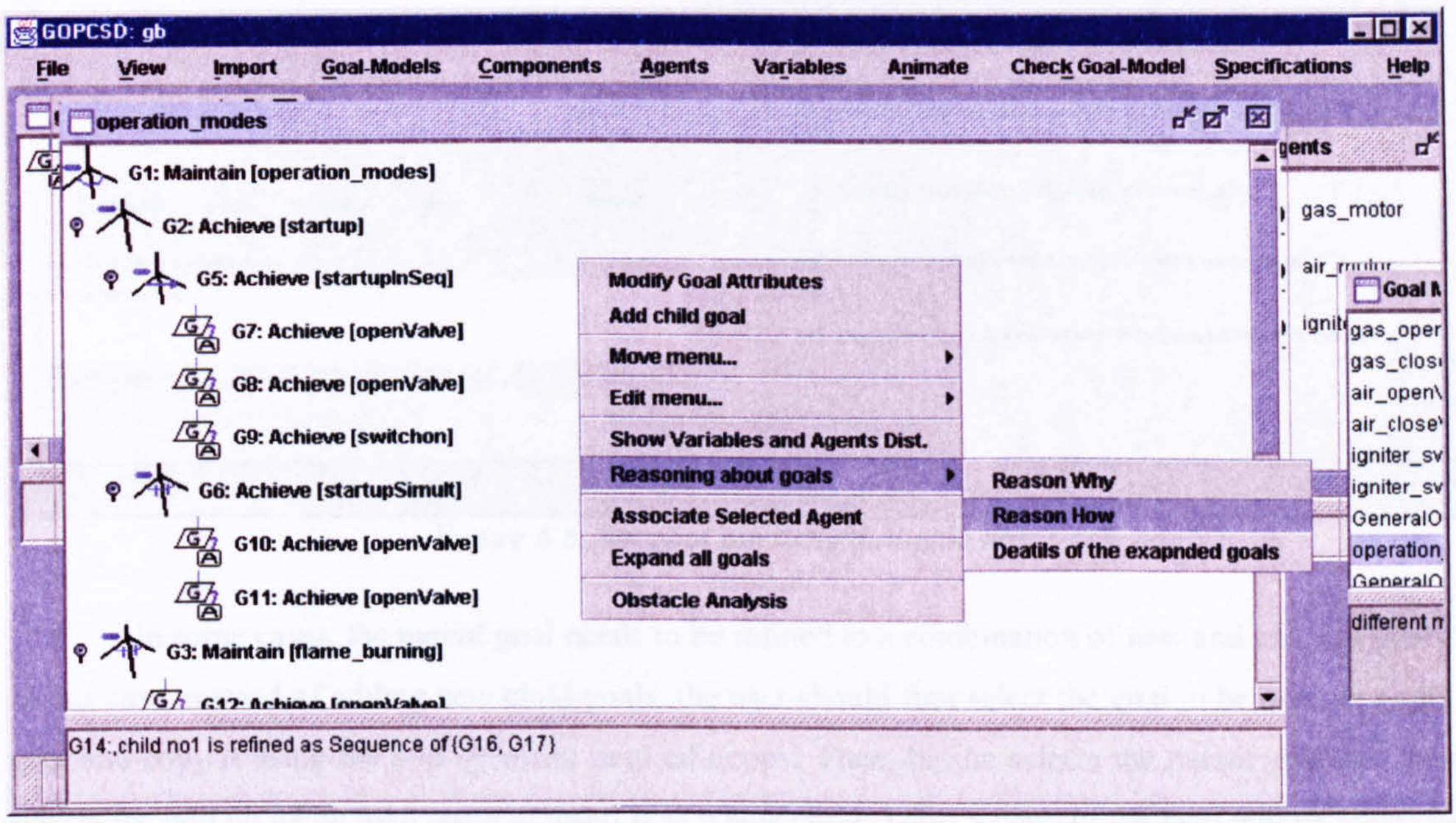


Figure 6.5, manipulating the different application's elements in the GOPCSD IDE

Similarly, the user can add/edit variables using the variable frame, agents using the agent frame, add/edit goals using the individual goal-model frames (as the one shown in figure 6.5) or add/edit goal-models using the goal-model frame.

6.2.3 Goal/goal-model Refinement (F3)

After the user adds the required elements of the application requirements, he/she has to integrate these separate goal-models into a single goal-model that is complete and describes the entire application. Thus, such a goal-model should have a root goal to describe the general aspects of the application. In addition, each terminal goal should be formally described and assigned to one of the

application's agents. The tool enables the user to refine the goals within their containing goal-model through the use of the pop up menu shown in figure 6.5. The refinement process can be accomplished in two steps: the first step is inside the parent goal where the refinement pattern can be changed, as shown in figure 6.6, where the user can activate the popup menu inside the goal-model containing the desired goal to be refined. Then, the user selects the modify goal-attributes sub-menu. A dialogue box containing the goal attributes will appear, as shown in figure 6.6 on the left.

The user can modify the refinement pattern of the selected goal by selecting the refinement attributes tab. Then, the refinement tab will appear, as shown to the right side in the figure, where he/she can find some guiding text written beside each refinement pattern to choose the suitable pattern. The second step is to use the pop up menu to add new sub-goals to the selected parent goal. The new sub-goals will be added by the default attributes; later, the user can select them and modify their attributes as will be explained later.

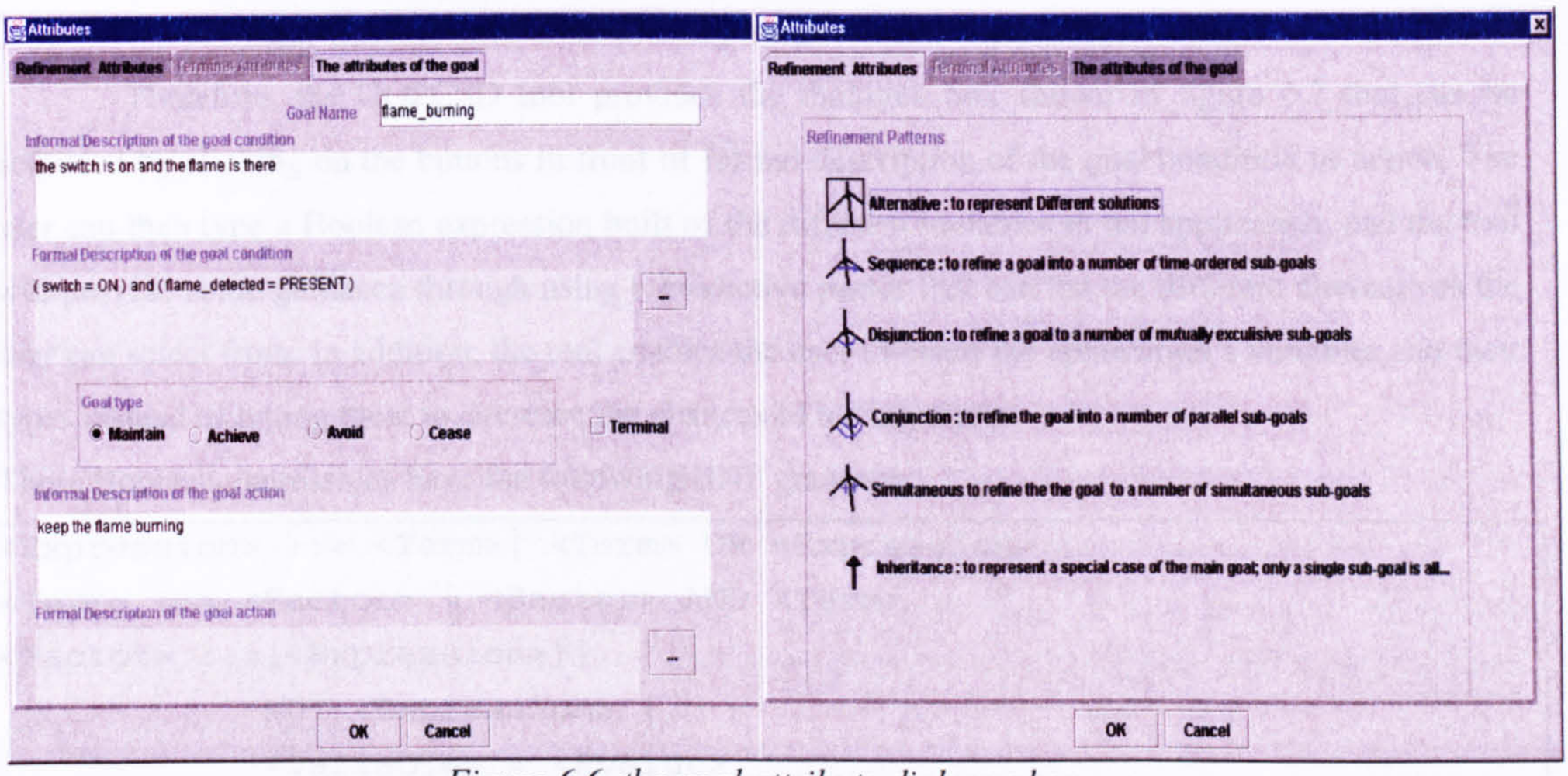


Figure 6.6, the goal attribute dialogue box

In some cases, the parent goal needs to be refined to a combination of new and existing goals; in this case, instead of adding new child goals, the user should first select the goal to be used as a sub-goal and copy it using the pop up menu item edit/copy. Then, he/she selects the parent goal and then selects the pop up menu item paste to insert it as a new child. Thus, a copy of the goal will be added as a sub-goal. The user can change the order of the sub-goals within the parent goal using the pop up menu item move goal/[move first| move previous| move next| move last] after selecting one of the sub-goals to move.

6.2.4 Goal/Goal-model Modification (F4)

The tool enables the user to edit the goal attributes after creating them; this can be accomplished in a similar way to refining a goal. The user selects the goal then edits goal attributes through the dialogue box shown in figure 6.6. The user can change the goal name, informal description, informal condition description, goal type and whether the goal is terminal or not by typing the appropriate description in the corresponding text boxes. Unlike these informal description

attributes, the formal condition and action of the goal need some guidance to reduce the chance of errors in writing them.

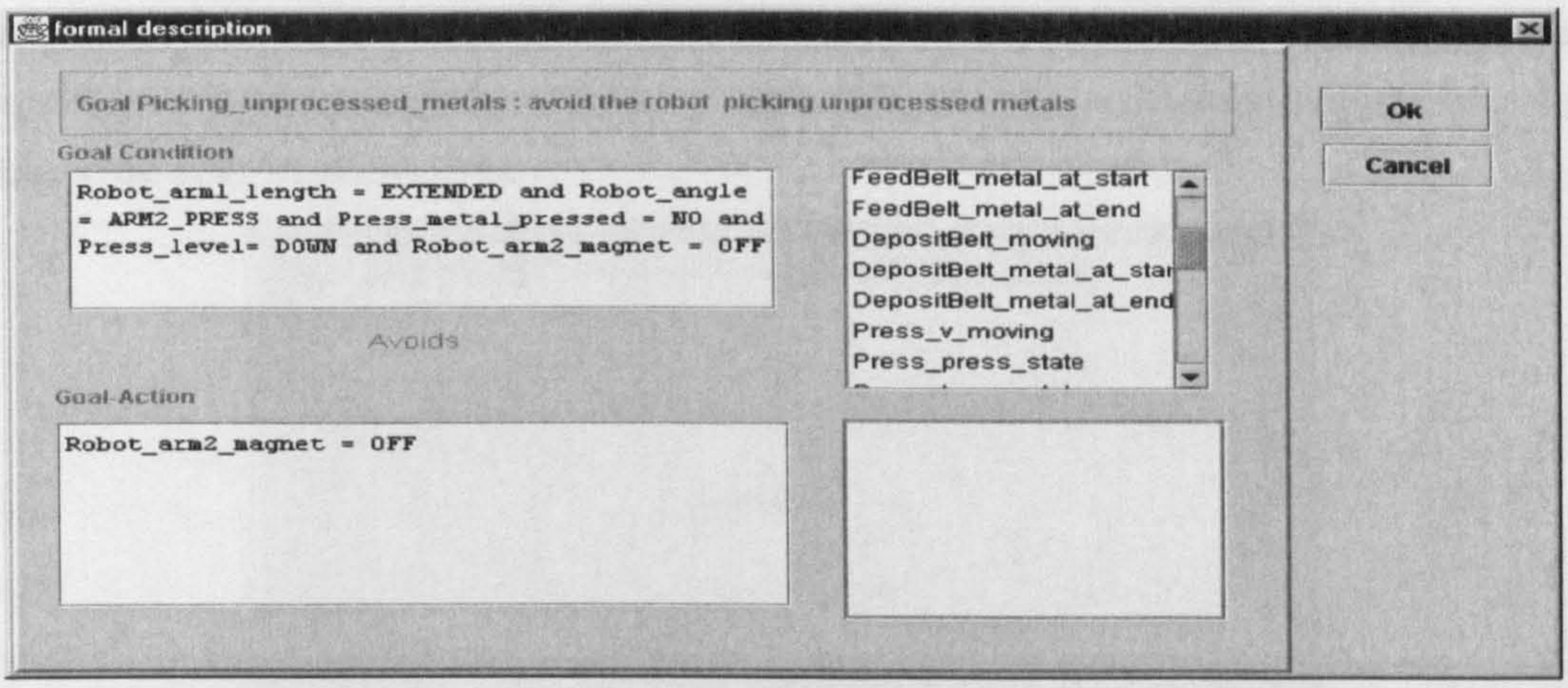


Figure 6.7, formal description of the goal condition and action parts

Therefore, the GOPCSD tool provides the dialogue box shown in figure 6.7 that can be activated by clicking on the buttons in front of formal description of the goal condition or action. The user can then type a Boolean expression built of the different variables in the application, and the tool will provide some guidance through using a predictive parser that can list the different alternatives the user can select from. In addition, the tool enables the user to insert the application's variables and their types instead of typing them to decrease the chances of having errors.

These Boolean expressions have the following BNF grammar:

```
<Expression> ::= <Term>| <Term> OR <Expression>
<Term> ::= <Factor> | <Factor> AND <Term>
<Factor> ::=(<Expression>)|
             NOT <Expression> |
             <Variable> = <Rvalue>
<Rvalue>::= <Variable> | <Value> | <num>
<Variable>::= identifier {found in the variable list}
<Value>::= identifier {found in the related types list}
<num>::= digits { string of digits }
```

where <Expression> is the start non-terminal, <Value> and <Variable> can be checked against the application variables list and their possible values.

We removed the left recursion from the above grammar to make it feasible for parsing, type checking, executing and evaluating the expressions. The precedence of the operators are designated as follows:

```
NOT
AND
OR
```

The *NOT* operator is unary while *AND* and *OR* operators are binary. These two binary operators are left associative. The parentheses can modify the precedence of the operator. The evaluation starts from the inner most.

By deciding that the goal should be a non-terminal goal, the user can activate the refinement tab by clicking on its name as shown in figure 6.6 on the right side. Otherwise, the user can edit the

terminal goal attributes by checking the terminal checkbox, then checking the terminal goal attributes tab, as shown in figure 6.8. The user can decide the terminal goal to be part of the application or the environment; this enables the user to add some environmental rules that can affect the animation and consistency of the application but will not be implemented in the specification stage because they belong to the external environment.

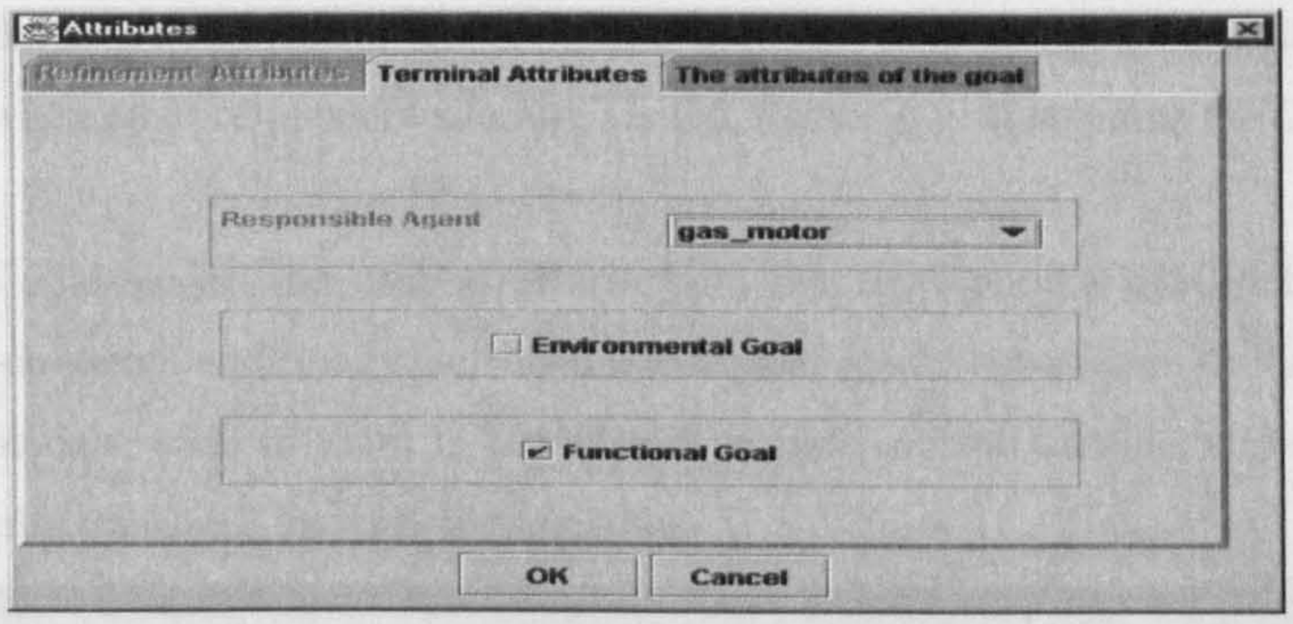


Figure 6.8, terminal goal attributes

6.2.5 Combining goals and goal-models (F5)

In many cases, after the user creates medium or low-level goals/goal-models, he/she needs to combine these with the existing goal-models by creating higher-level goal-models to combine them. Therefore, the tool enables the user to combine separate goals/goal-models as sub-goals of a new parent goal. This combination can be accomplished as follows: the user creates a new goal-model within which to combine the sub-goals; then the user selects each of the sub-goals in turn and copies them to the application clipboard using the goal-model pop up menu; then, selects the root goal of the newly created goal-model and selects paste as a new child. This will copy each goal as a sub-goal to the new goal-model's root goal. Afterwards, the user can get rid of the separate goal-models or keep them for other purposes.

6.2.6 Agent Assignment (F6)

As mentioned earlier, each terminal goal has to be assigned to an agent to accomplish it. The agent assignment process arises when deciding on the goal to be terminal and then selecting the terminal attributes tab, as shown in figure 6.8. The user selects the appropriate agent from the combo box. Alternatively, without going through the goal-details, the user can select the appropriate agent from the application's agents list, and then select the required terminal goal, activate the pop up menu and finally, select the assign selected agent sub-menu. If the user by mistake attempts to assign the agent to a non-terminal goal or to an already assigned goal, the tool will take the appropriate actions. Moreover, after the user completes the goal-model, a general check can be performed that ensures each terminal goal has been assigned to an agent and also ensures the uniqueness of variable control so that none of the application's variables is controlled by more than one agent, at different terminal goals.

6.2.7 Splitting Alternatives (F7)

The first step before judging or checking a solution is to split the compound goal-models by cutting the goal-model at the alternative refinement sites. As explained earlier in section 5.5, we used

the goal refinements semantics to derive the splitting algorithm. Based on the *split* function and the corresponding rules 5.34, 5.35, 5.45, 5.36, 5.37, 5.38, 5.39, 5.40, 5.41 and 5.42, the implemented bottom up algorithm splits the compound goal-models at the alternative refinement sites. The algorithm starts at the terminal-goal level and ascends the goal-model tree, propagating the possible alternative goal-models for each goal; the possible solutions will be represented as a list of goal-model trees. When an alternative refinement site is visited, the main goal combines the different lists of its sub-goals, while when other refinement sites are visited, the main goal acquires the Cartesian products of its sub-goals.

To split a goal-model, the user selects it from the application's goal-model list, and then activates the pop-up menu, and finally, selects the split goal-model sub-menu. This action generates a number of goal-models; each of them is considered an independent solution to be tested, checked, animated, and finally translated into a B specification.

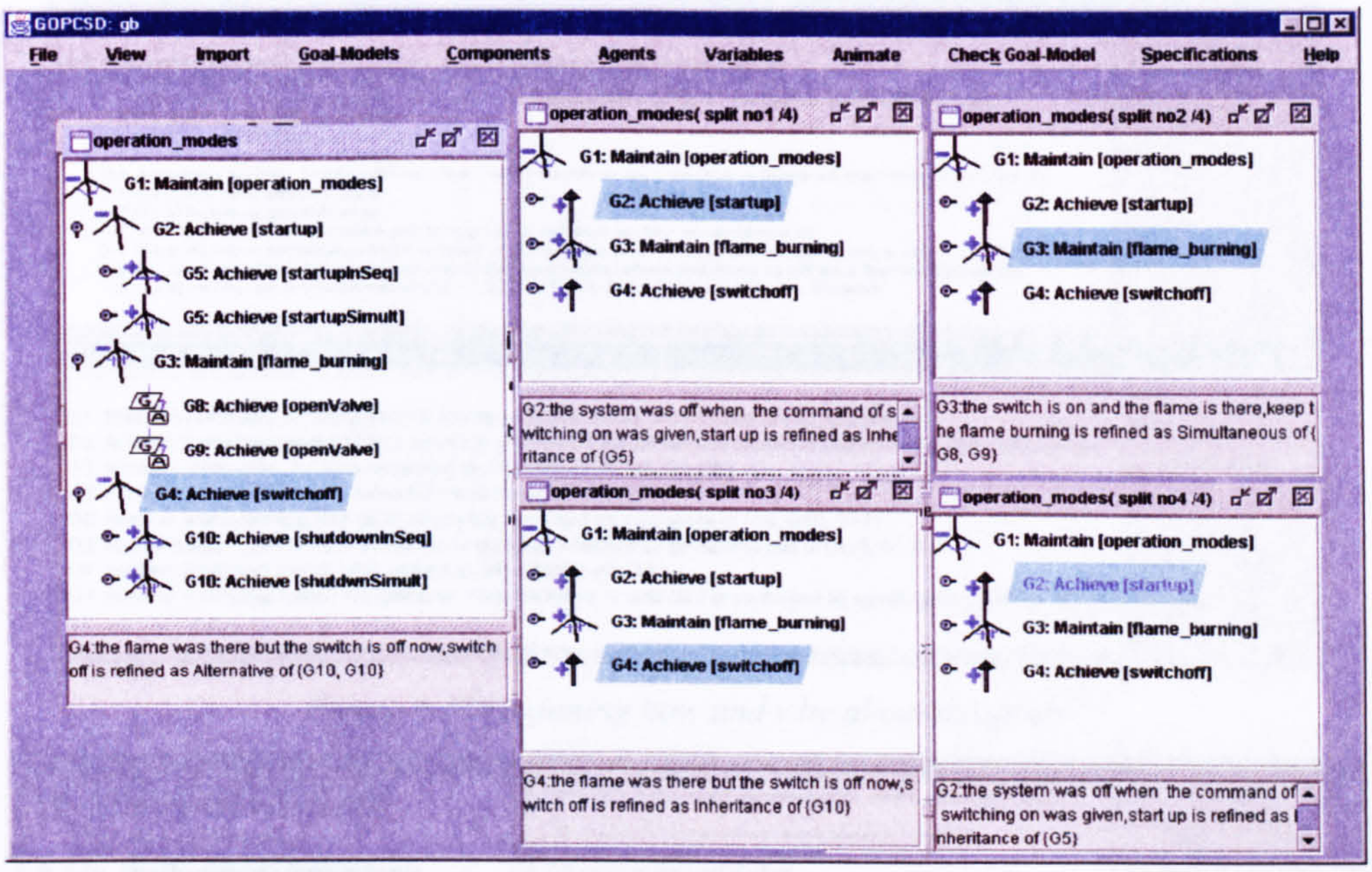


Figure 6.9, splitting compound goal-models

Figure 6.9 shows an example of splitting a compound goal-model (to the left), named *operation–modes* that has two goals refined as alternative (G2 and G4); each of them has two sub-goals. This splitting process results in generating four final versions of goal-models (to the right) that can be checked, animated and finally translated into a B specification.

6.2.8 Reasoning and Investigation Utilities (F8)

The GOPCSD tool enables the user to reason how and why about the different goals of the goal model. Reasoning how to achieve one goal lists the sub-goals of this goal and their sub-goals; while reasoning why to achieve one goal ascends the goal-model level by level through listing the details of the ancestor goals. In addition to how and why reasoning, the tool enables the user to reason how about the entire goal-model but with the ability of hiding some details of the sub-goals; this can be useful for large goal-models and especially, when the user has already reason about a sub-goal (the user may hide this sub-goal). Thus, the reasoning utility can be regarded as an early level of checking

to validate parts of the goal-model, on the one hand; on the other, it can be used to guide the user to elicit new goals to complete the construction of the goal-model either upward answering why or downward answering how [Van Lamswerde et al. 91]. Figure 6.10 shows samples of the goal reasoning utility. Another utility that cannot be easily classified to phase I or phase II is highlighting the goals containing a specific variable or the goals that are affected by a specific agent. This utility is similar to dependence graphs, or tracing utilities, which can be helpful to judge variable and agent coupling and the ability of moving goals within the goal-models. Figure 6.11 shows an example of highlighting goals that contain variable igniter_state of a gas burner application.

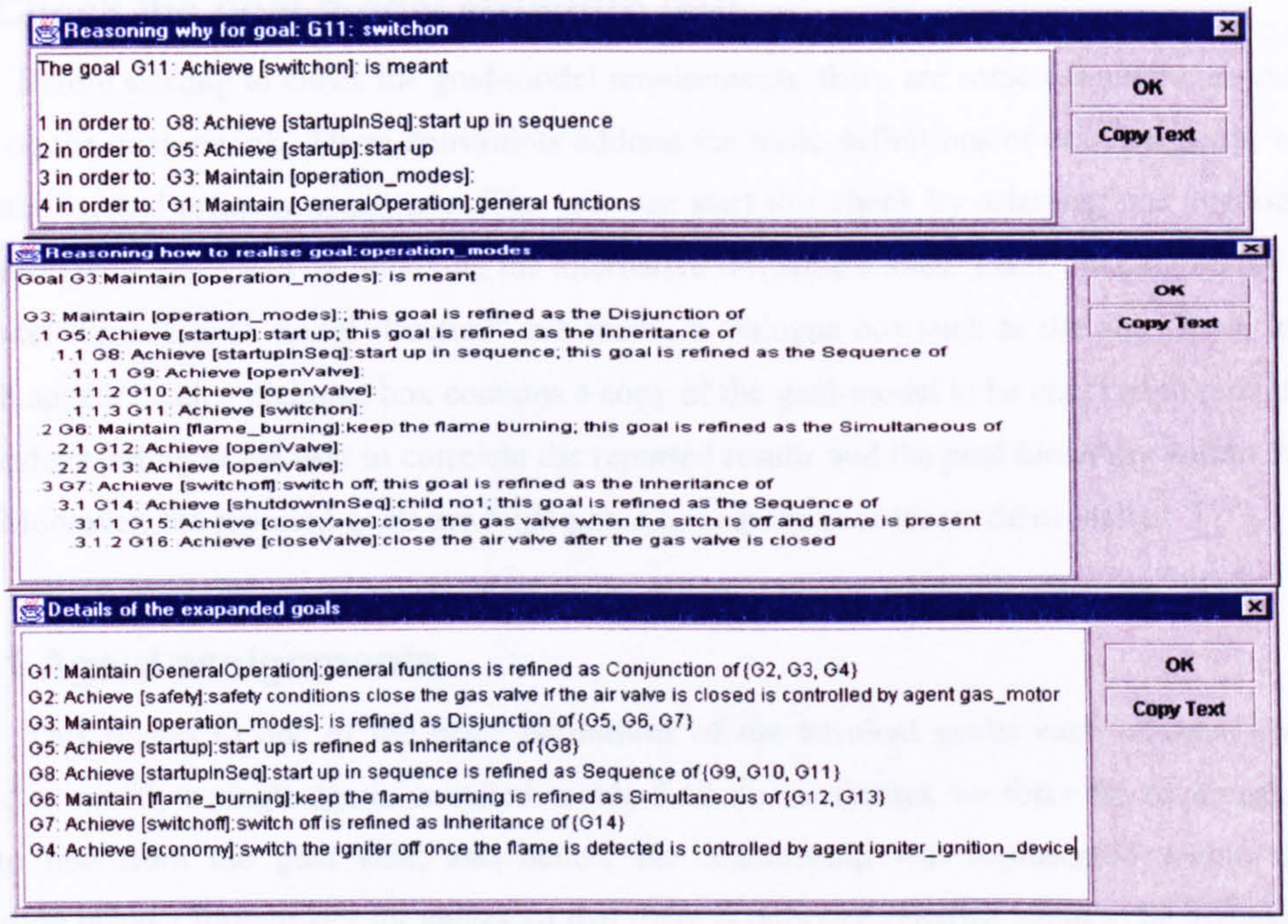


Figure 6.10 reasoning how and why about the goals

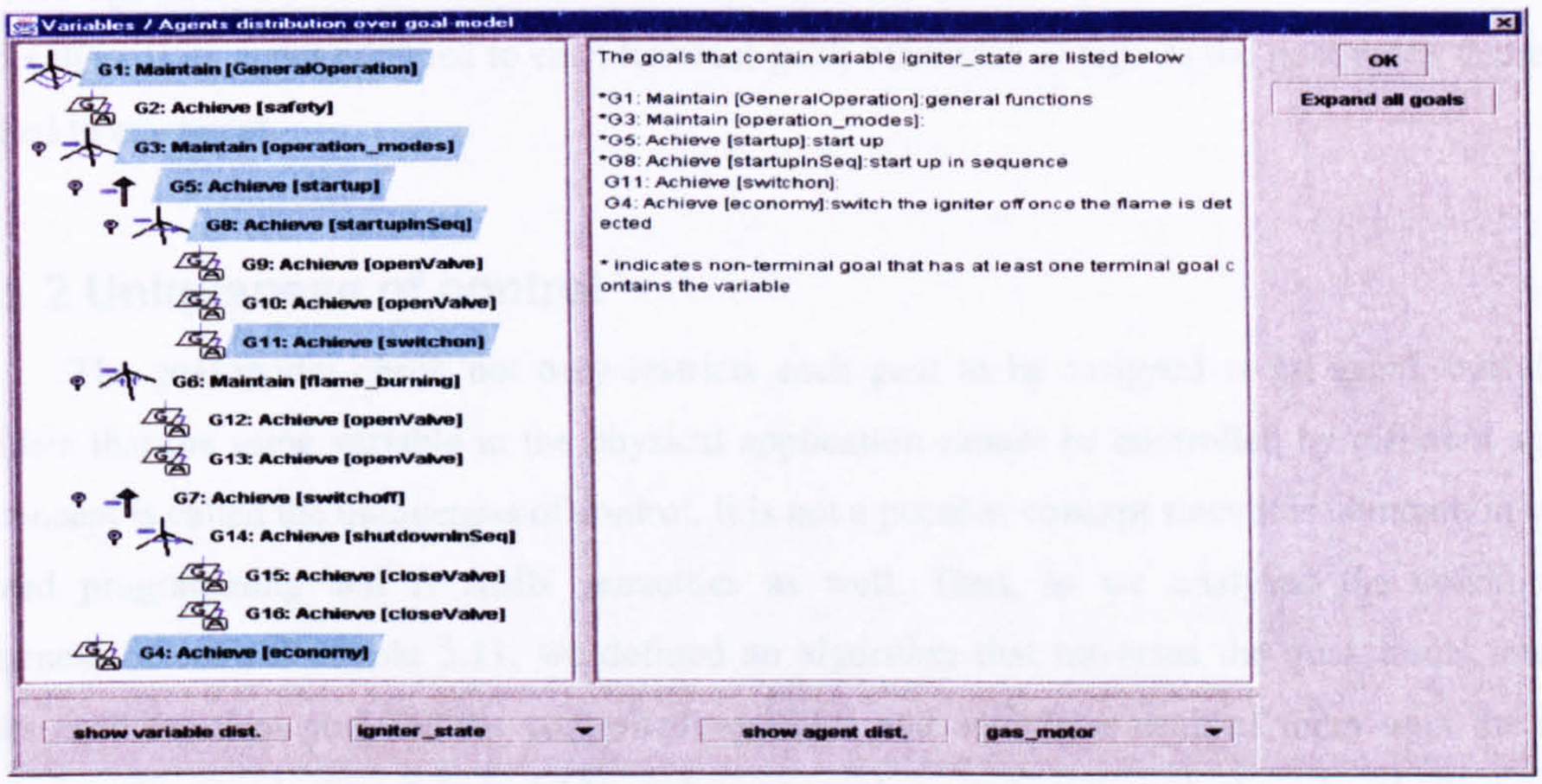


Figure 6.11, highlighting goals containing a specific variable

6.3 Requirements Checks and validation (Phase II)

Although the GOPCSD tool guides the user during the first phase to construct correct goal-models, the development environment cannot be considered strong and useful unless it expects some

requirements problems and hence, provides enough checks, tests, validations, advice and suggestions to remove these problems. The GOPCSD tool enables the user to check the requirements' completeness, consistency, reachability, obstruction, and validation. In each case, the tool provides guidance for the user to modify the goal-model in order to get rid of detected problems, which may not be discovered later, or discovered but with a higher cost and effort. In the following sub-sections, we describe in detail the second development phase, which consists of six checks enabling the user to modify the requirements before the automatic translation to B AMN machines takes place.

6.3.1 Check the goal-model structure (F9)

Before starting to check the goal-model requirements, there are some essential constraints we enforce on the goal-models. These constraints address the basic definitions of terminal goals, variable controllability, and refinement patterns. The user can start this check by selecting one version of the final goal-models generated by removing the alternative refinement sites. Then, clicking on the "check goal-model"/"check-goal model structure" sub-menu, a dialogue box such as the one shown in figure 6.12 will appear; such a dialogue box contains a copy of the goal-model to be checked to provide some visual understanding to the user to correlate the reported results and the goal hierarchy within the goal-model. Moreover, the reported goals are highlighted to help the user locate them easily.

6.3.1.1 Agent assignments

This relates to one of the basic definitions of the terminal goals; each of them should be assigned to exactly a single agent, as stated in rule 5.10. In our design, we force the relationship to be many to one from the goal side; and hence, the relationship was represented within the goal information using a foreign key for agents so that there is only one possible agent to be assigned to the goal (see appendix A, section A.3, for more details). So, the check traverses the goal-model and ensures there is an agent assigned to each terminal goal; otherwise, it reports the goal index that is not assigned to any agent.

6.3.1.2 Uniqueness of control

The goal-model check not only restricts each goal to be assigned to an agent, but also it considers that the same variable in the physical application cannot be controlled by different agents; this concept is called the uniqueness of control. It is not a peculiar concept since it is common in object oriented programming and B AMN semantics as well. Thus, as we analysed the condition of uniqueness of control in rule 5.11, we defined an algorithm that traverses the goal-model tree and checks each terminal goal for the controlled variables and associates each of them with the agent assigned to this terminal goal (rule 5.12); it then proceeds to the next terminal goal and if a previous variable, is contained in this goal, which is controlled by a different agent, the algorithm reports the terminal goal, the variable and the different agents so that the user can modify the agent assignments.

6.3.1.3 Simultaneous refinement constraints

This check ensures that each simultaneous refinement site has only sub-goals of type terminal goal. We implemented an algorithm based on rule 5.33 to traverse the goal-model tree and consider the simultaneous refinement sites, and hence, check whether each sub-goal is a terminal goal or not (by examining whether it has sub-goals or not). In the case of having a non-terminal sub-goal, the user will be guided to change the refinement pattern to conjunction; or, alternatively, he/she can change the goal-model structure.

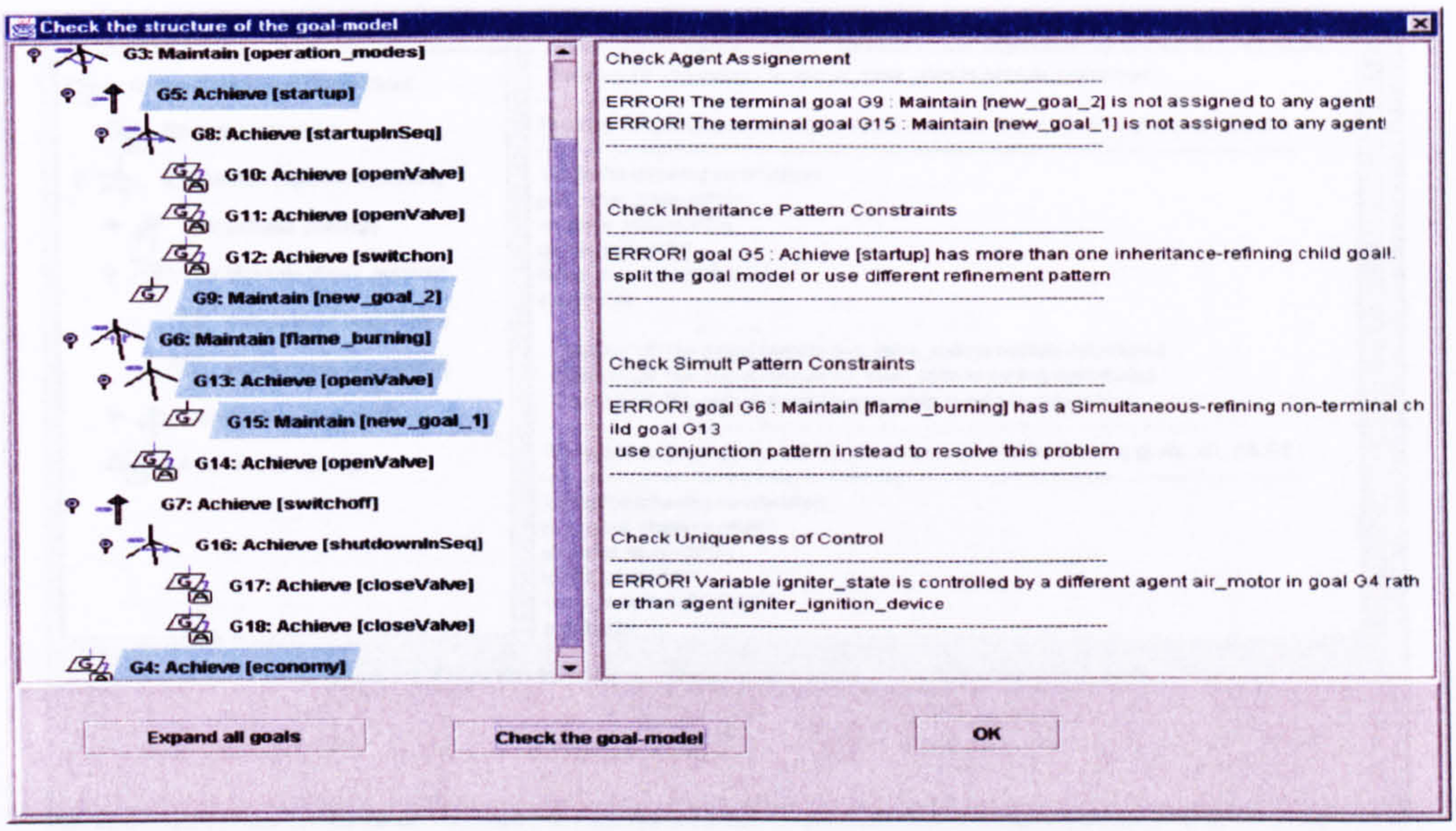


Figure 6.12, check the goal model for agent assignment and refinement restrictions

6.3.1.4 Inheritance refinement constraints

To eliminate inconsistency or ambiguity between the goals of a single goal-model, we restricted the number of sub-goals of the inheritance refinement pattern to be one, as described in rule 5.32. The algorithm ensures this restriction through checking that each inheritance refinement site has exactly a single sub-goal. In the case of violating this restriction, the parent goal will be reported to the user; he/she may either split this goal model or change the refinement type to a more suitable pattern.

6.3.1.5 Alternative refinement constraints

Unlike the constraints of the simultaneous and inheritance refinement pattern, the alternative constraints is not an error but a kind of warning to inform the user that this goal-model needs splitting before it can proceed to the following check stages. The algorithm basically traverses the goal-model tree and reports any non-terminal goal that has alternative refinement type.

6.3.2 Completeness Analysis (F10)

Completeness analysis can reveal situations, which are not considered by the systems engineer, as in [Leveson Y2K]. In the GOPCSD tool, we follow the definition of completeness in

[Davis 93]. A completeness check on the condition that for each combination of the application variables there should be a defined action(s) to be taken, and these actions determine the output variables, as we formally expressed in rule 5.53.

Instead of considering all possible combinations of the application variables, we consider each output variable separately; only the effective variables that control this output variable will be considered. This shortens the time required to perform the check, especially in large-scale systems. Moreover, the cases reported will be simpler rather than listing all application combinations.

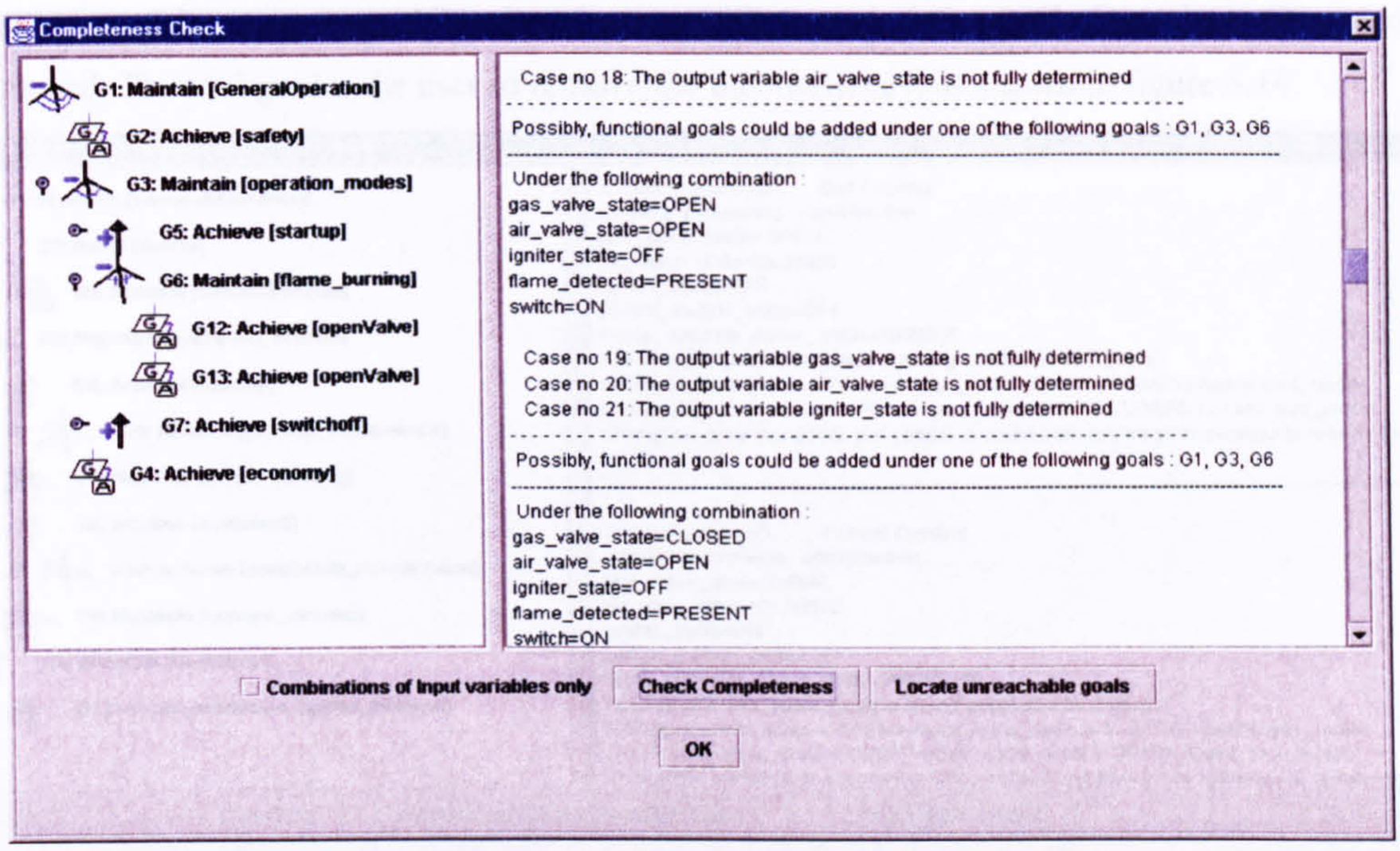


Figure 6.13, check the completeness of the goal-model

Some incompleteness can occur as a result of ignoring unexpected variable combinations or ignoring variable combinations under specific situations, which the systems engineer considers to be impossible due to some knowledge about the environment. However, the systems engineer's decisions should be taken after full awareness of these situations.

We implemented a completeness check that reports any output variable that is not fully determined by the inputs. The output variable is considered fully determined under variable combinations when the assigned value after one execution cycle is always the same independent from its initial value. In this sense, the completeness check should be performed on a consistent goal-model. Figure 6.13 shows the corresponding dialogue box as a result of performing the completeness check. The dialogue box shows the checked goal-model. In the case of discovering incompleteness, the variable combination, undetermined output variable modifications and suggestions will be reported separately in each case to guide the user to complete the requirements.

6.3.3 Goal-Conflict Check (F11)

After ensuring the user provided a valid goal-model and completed the requirements, it is important to ensure that the different goals of the goal-model are consistent. Otherwise, possible deviations and unexpected scenarios may take place during run-time. As descried in chapter 3 (rules

3.1 and 3.2), the goal-conflict arises when two or more goals prescribe the performance of inconsistent actions under the same conditions [Easterbrook 94, Van Lamsweerde et al. 98b].

Goal conflicts are checked by comparing the conditions for goals that assign different values to the same variables. We reformulated 3.1 and 3.2 to 5.49 in order to discover goal pairs that can cause conflict during run-time.

For example, in a gas burner system, one goal, like keep the flame burning, that requires the air and gas valves to be kept open, may conflict with some safety goal restricting the gas valve to open only when the air valve is open. The conflict will be reported only if the pre-conditions of the two goals are true at the same time. After detecting a conflict, one of the goal's formal conditions can be strengthened. The tool guides the user to remove the inconsistency, as shown in figure 6.14.

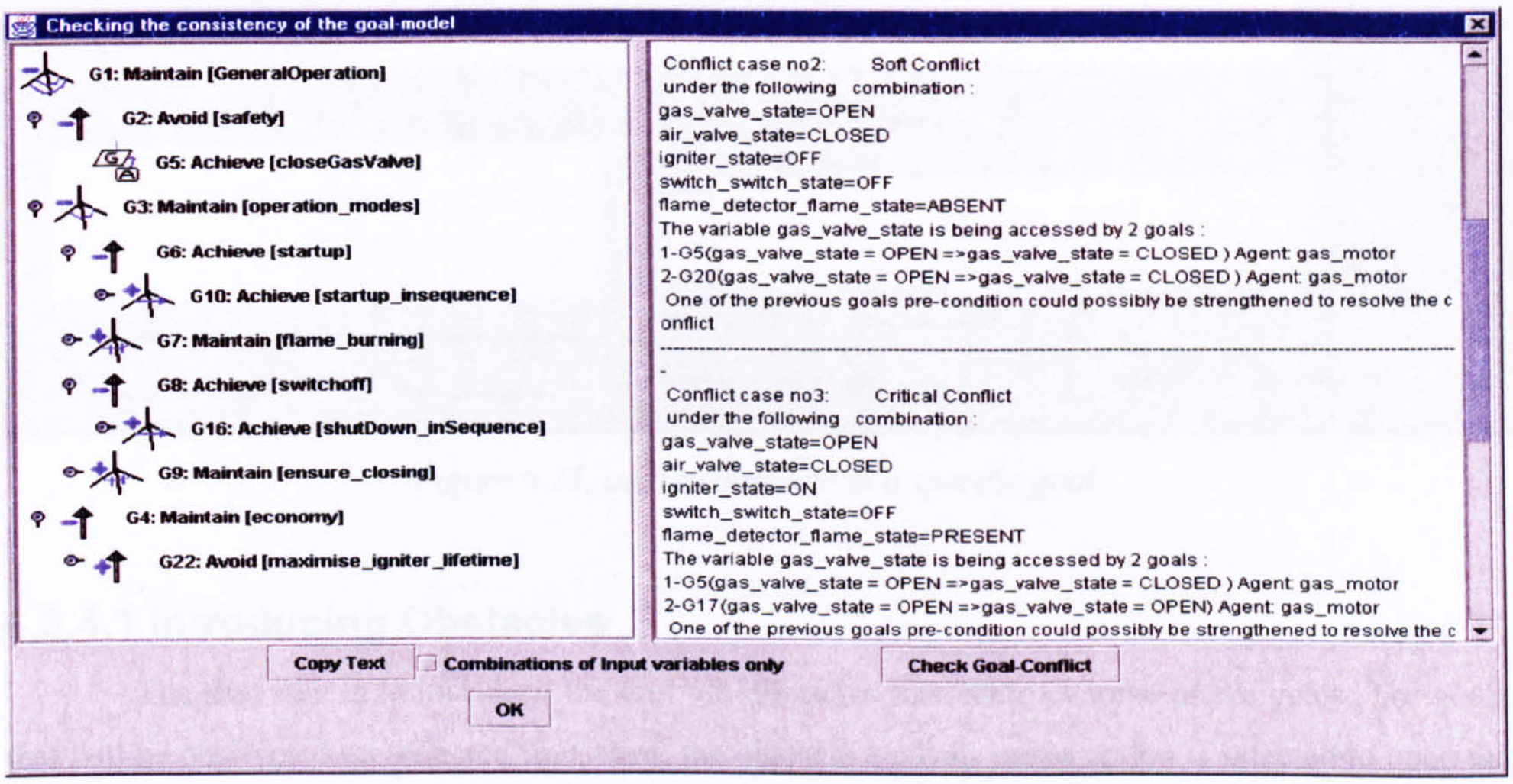


Figure 6.14, discovering goal-conflicts

As noticed from the figure, when a number of different terminal goals simultaneously assign values for output variables, the conflict is reported; however, the tool has defined two types of conflicts. If the values assigned by the different goals are exactly the same value, the conflict is called "soft" because re-ordering the execution of the inconsistent goals does not have any impact on the over all performance. But, if the assigned values to the same output variable differ from one terminal goal to another, the conflict is called "critical"; and hence, the tool provides suggestions for the user to get rid of these critical conflicts.

Furthermore, the tool enables the user to choose only the critical conflict cases, and discard the soft conflicts. This can be considerably helpful, especially, when large number of conflict cases is reported either due to problems in requirements or large-scale system.

6.3.4 Obstacle Analysis (F12)

Obstacles are a sequence of events that can happen during application run-time and can obstruct some of the goals from being achieved. Obstacles themselves can be either simple or regarded as a refinement hierarchy in which the compound obstacle is composed of sub-obstacles. There are existing methods to generate and identify obstacles for a given goal, as presented by Van Lamsweerde

and Letier in [Van Lamsweerde et al. 98a], where they suggested methods to modify the goal-model after detecting obstacles, depending on the type of the obstacle. Obstacles can be formally analysed using rules 3.3, 3.4, and 3.5. The user can consider the obstacle analysis as one way to deepen the application's scope by considering environmental conditions which were not considered before and which usually affect the application's operation. The user can perform the obstacle analysis in three steps as follows.

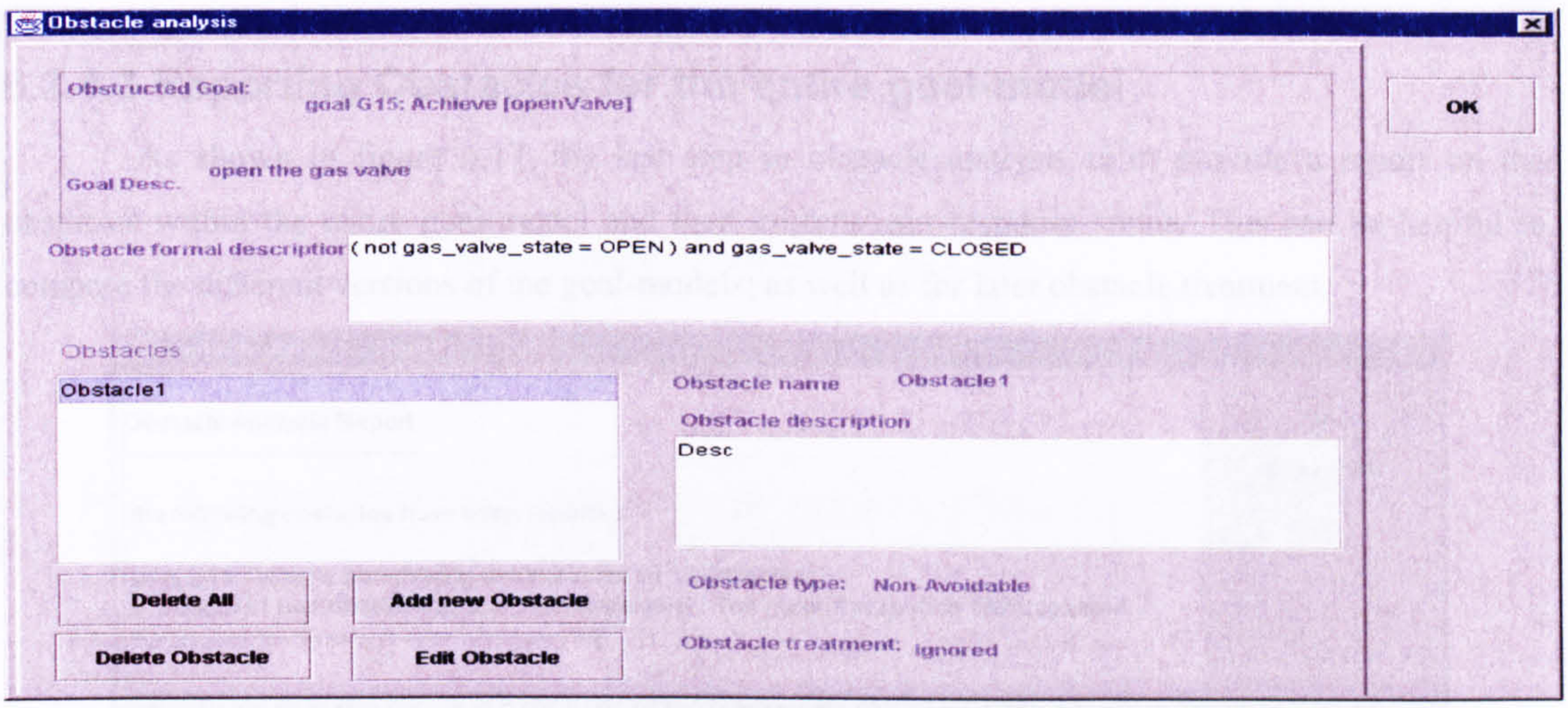


Figure 6.15, adding obstacle to a specific goal

6.3.4.1 Introducing Obstacles

The first step is to introduce the tool via obstacles that obstruct some of the goals. The goals that will be obstructed are selected first; then, the obstacle analysis menu option is selected to open the dialogue box shown in figure 6.15. As shown in the figure, the user can see the negation of the goal's formal description that is the description for the obstacle to occur, and then he/she can think of different reasons why such an obstruction can happen during run-time. Each obstacle will be recorded with different a name and description to be accessed later. For effort reduction, it is recommended to consider only the obstacles of the terminal goals.

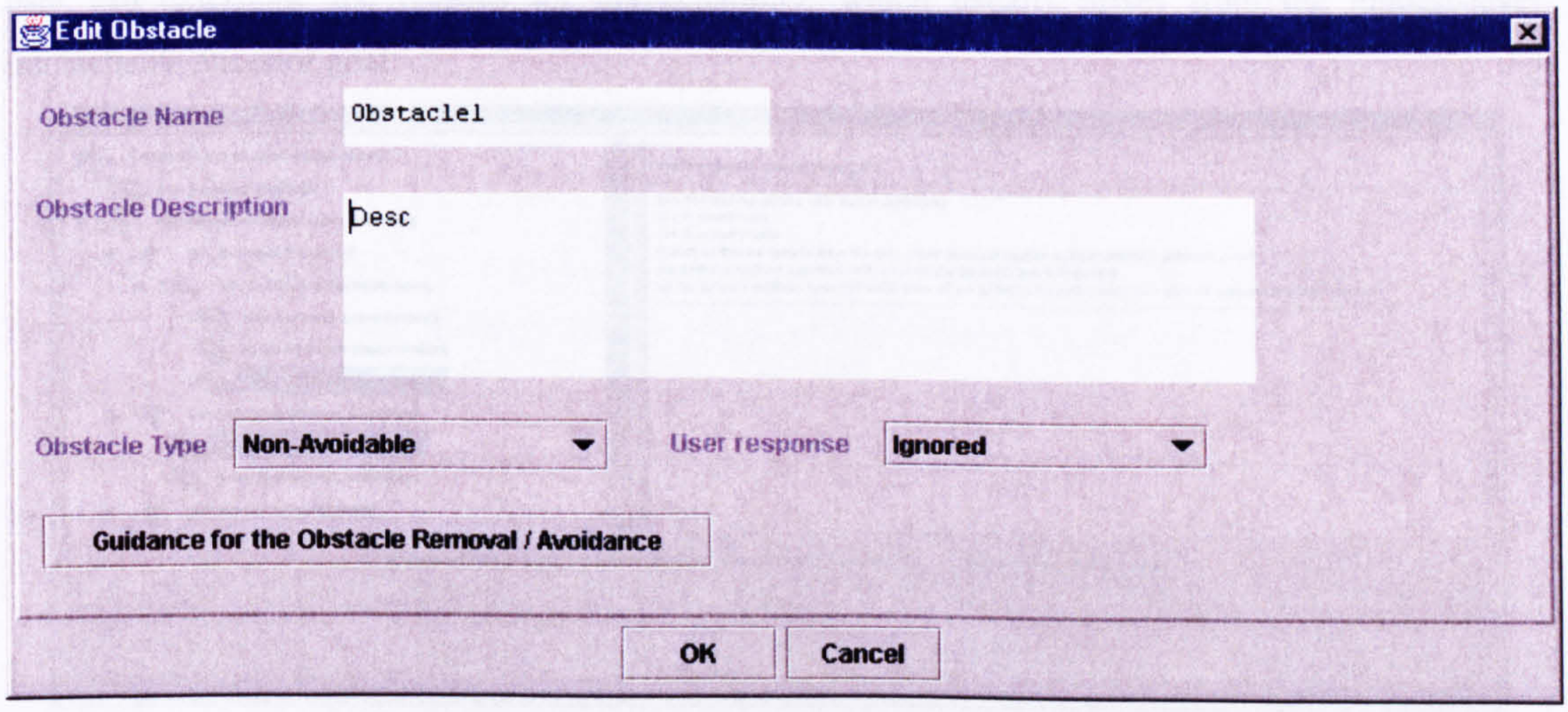


Figure 6.16 modifying the obstacle status

6.3.4.2 Editing Obstacle Status

The second step is to change the obstacle status (user response to it) by first selecting the desired obstacle from the Obstacles list in figure 6.16, and then clicking on the edit obstacle button. Each obstacle can be avoidable, amendable, non-avoidable or removable. The GOPCSD tool provides some guidance to the user to inform him/her how to get rid of/avoid/amend the effect of the obstacle. Accordingly, he should update what is the current response for this obstacle.

6.3.4.3 Reporting Obstacles for the entire goal-model

As shown in figure 6.17, the last step in obstacle analysis is to provide a report on the obstacles within the entire goal-model and their current user-response status. This can be helpful to compare the different versions of the goal-models; as well as for later obstacle treatment.

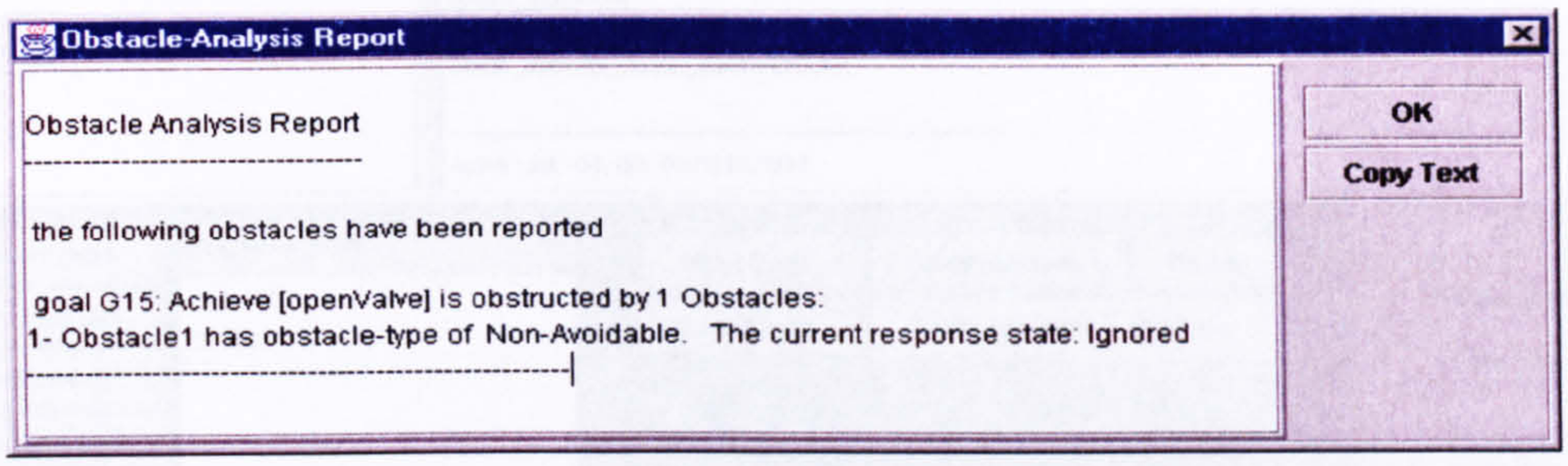


Figure 6.17, Obstacle report

6.3.5 Detecting unreachable goals (F13)

Another important check that is helpful to amend the requirements is detecting unreachable goals. The user usually errs in specifying the conditions of the some of the goals or locates them in inappropriate locations where they will never be activated. Thus, the goal-model can be valid, complete and consistent but some of its goals can be unreachable. To discover these unreachable cases, we implemented an algorithm based on rules 5.50, 5.51 and 5.52. Such an algorithm traverses the goal-model tree and ensures that each goal's accumulated pre-condition is not always false. Using the fact that if the parent goal is unreachable all of its sub-goals will be unreachable (rules 5.51 and 5.52), the user can determine the reasons for unreachability, which usually arises from the highest-level unreachable ancestor goal.

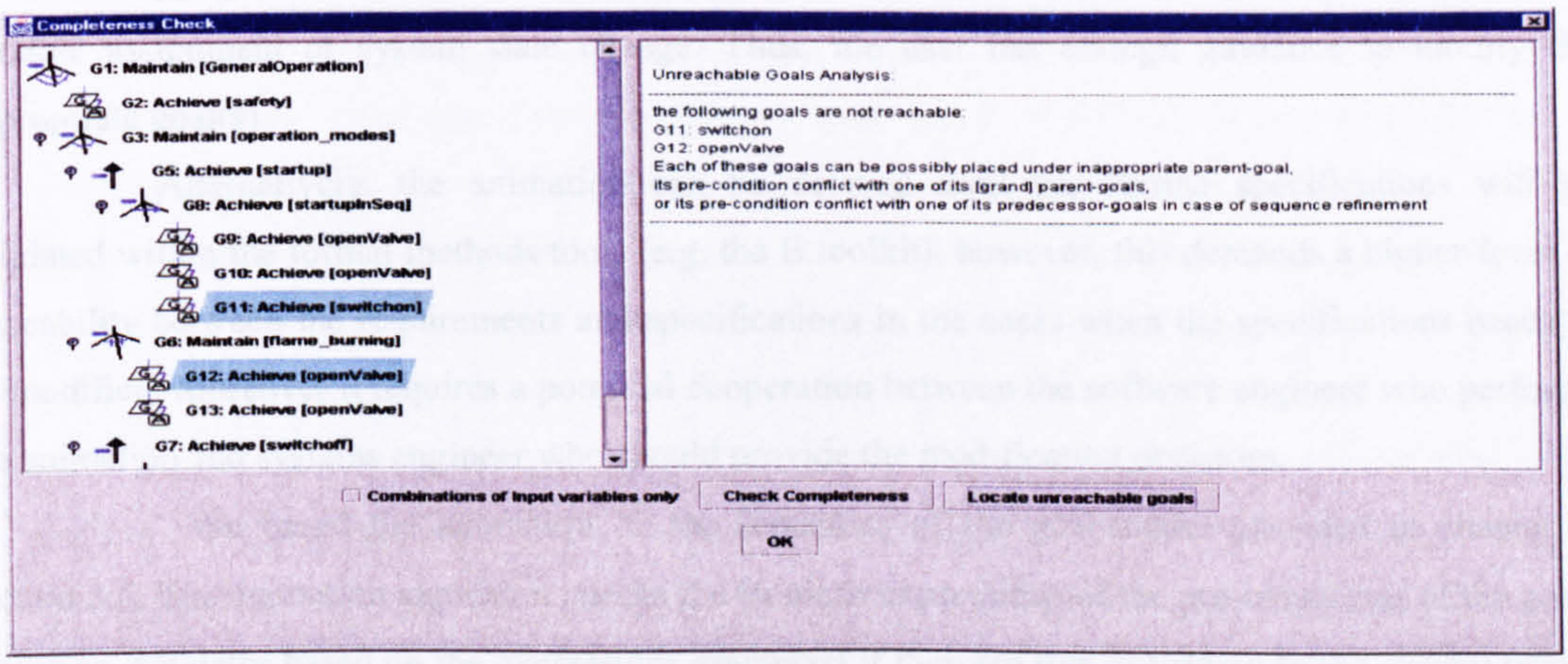


Figure 6.18, detecting unreachable goals

Figure 6.18 shows a dialogue box that reports unreachable goals. The tool discovers such cases and highlights the unreachable goals within the goal model and reports them to the user in the text panel with some suggestions and possible predictions about why these goals are unreachable.

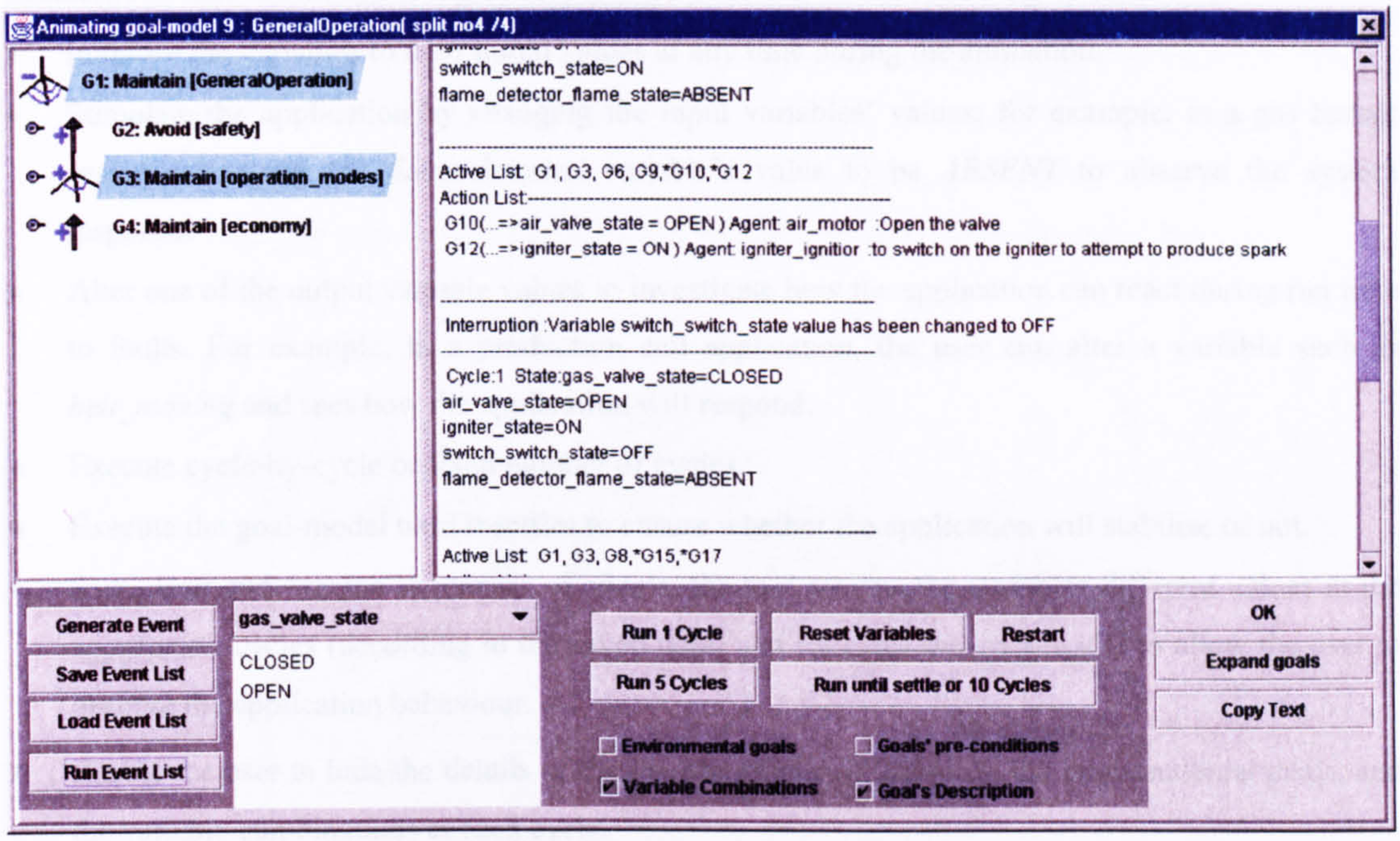


Figure 6.19, requirements validation

6.3.6 Requirements Animation (F14)

After performing the completeness, conflict, reachability and obstruction analyses, and modifying the goal-model to amend the requirements, the systems engineer usually still needs to validate the requirements. The need of validating the requirements has two main sources: after passing all these check stages, they could possibly be different from the initial ones and the second reason is that the systems engineer may state some incorrect requirements that could not be discovered from completeness, consistency or reachability tests.

Such validation should emulate the execution of the controller during run-time and works as a goal-model debugging tool. It should always inform the user about which goal(s) are responsible for variable assignment or system state change. Thus, the user has enough guidance to modify the appropriate goal(s).

Alternatively, the animation can be delayed until the formal specifications will be validated within the formal methods tools (e.g. the B toolkit); however, this demands a higher-level of traceability between the requirements and specifications in the cases when the specifications needs to be modified. Moreover it requires a potential cooperation between the software engineer who performs the animation and systems engineer who should provide the modification decisions.

We based the animation on the semantics of the goal-model provided in chapter 5, section 5.5. The animation algorithm checks the Boolean expressions of the pre-conditions of the goals (using an evaluator based on the expressions grammar) if they are true according to the current values of the application's variables, it adds them to the active list, highlight them, and then proceed to their

sub-goals. When the algorithm reaches a terminal goal, if the pre-condition is true, the algorithm executes the goal-action and updates the controlled variables. Thus, as shown in figure 6.19, using the animation dialogue box controls, the user can visually, notice the active branches from one cycle to another. The user interaction is highly respected; he/she can do the following actions:

- Initialise the variables to their initial values at any time during the animation.
- Stimulate the application by changing the input variables' values; for example, in a gas burner system, changing the *flame-detected* variable's value to be *ABSENT* to observe the system response.
- Alter one of the output variable values to investigate how the application can react during run time to faults. For example, in a production cell application, the user can alter a variable such as *belt_moving* and sees how the application will respond.
- Execute cycle-by-cycle or fixed number of cycles.
- Execute the goal-model until it settles to ensure whether the application will stabilise or not.
- Save, load and execute sequences of events; the tool assigns the variables different values at the appropriate cycles (according to the saved data) and executes the goal-model to allow the user to observe the application behaviour.
- Enable the user to hide the details of the pre-conditions of the goals, the environmental goals, and the variable combinations at each cycle.

6.3.7 Checks dependency

Since the tool provides a considerable number of checking and validation analyses, there should be user-guidance for the order of the different checks the developed goal-model undergoes until it reaches a state in which it is ready to be translated to B specifications. Figure 6.20 represents these stages before and after performing the different checks and validation. The states of the state transition diagram are described as follows:

- State St1 (Developed goal-model): the initial state. The goal-model reaches this state after it has been created or edited (changing its formal structure).
- State St2 (Split goal-model): the state of the individual goal-models after splitting the compound goal-model containing alternative refinement sites.
- State St3 (Valid structure goal-model): this state can be reached directly from the initial state by checking the goal-model and ensures that its structure does not violate the constraints or by splitting the goal-model and then checking each of the individual solutions.
- State St4 (Consistent goal-model): this state can be reached after performing the goal-conflict analysis for a goal-model with a valid structure (state S3). In addition, the goal-model should either be proved to suffer no inconsistency or to be agreed by the user.
- State St5 (Complete and reachable-goals goal-model): this state can be reached if the goal-model with a valid structure has correctly passed the completeness and reachability tests or the user agrees to consider it complete. For example, the user can ignore the incompleteness cases because they are impossible.

- State St6 (Validated goal-model): this state will be reached when the goal-model formal requirements have been animated and approved by the user.
- State St7 (Obstacle analysed goal-model): this state will be reached when the user performs the obstacle analysis for the goal-model's goals.
- State St8 (Ready to generate B specification): this state can be reached when the user has checked the structural correctness of the goal model, performed the goal-conflict analysis, performed the reachability and completeness analysis, and finally animated the goal-model. The obstacle analysis is not an obligatory step before translating the goal-model into formal specifications.

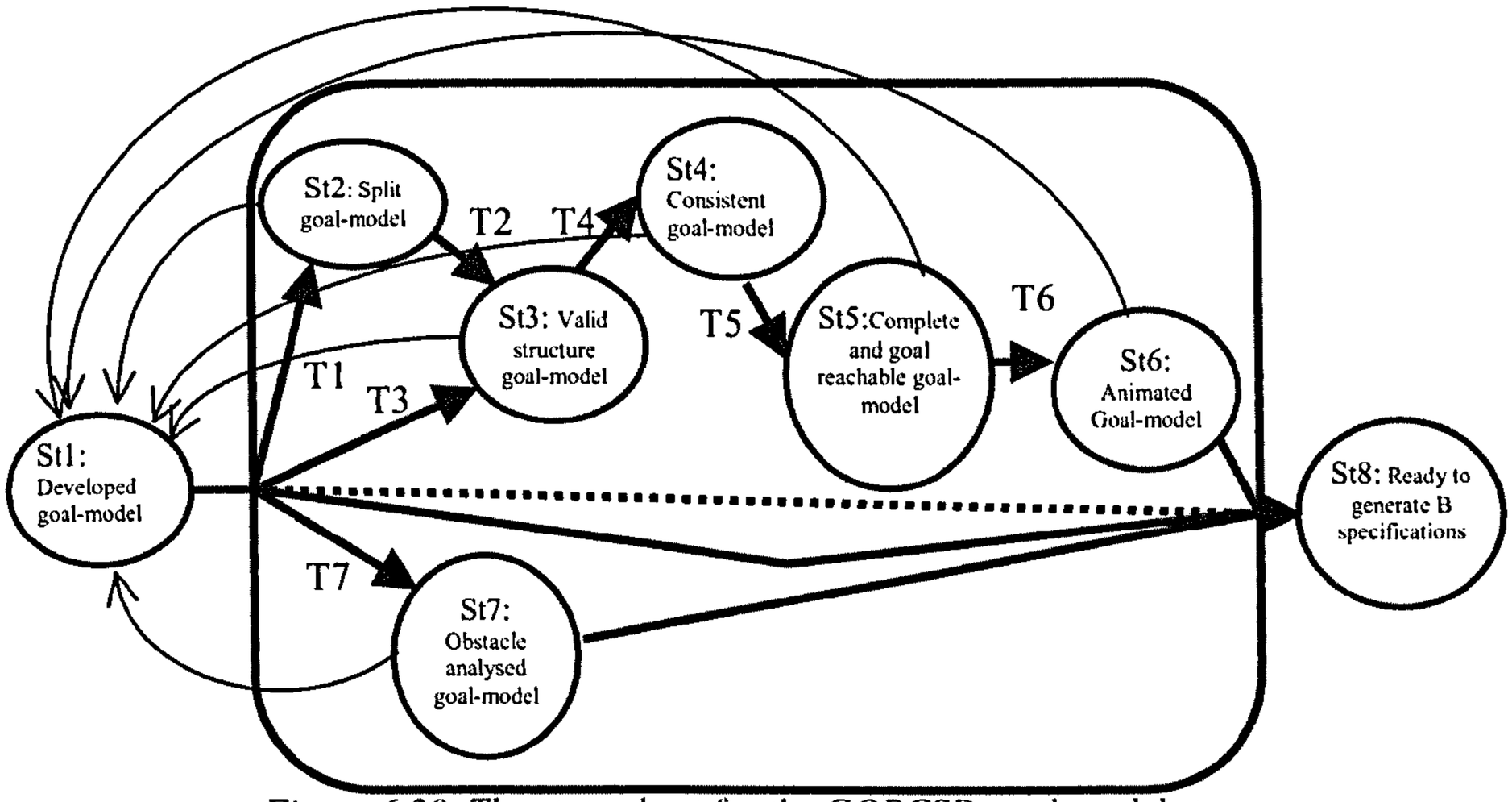


Figure 6.20, The state chart for the GOPCSD goal-model states

And the transitions are defined as follows:

- Transition T1: splitting the compound goal-model
- Transition T2: performing the goal-model structure check on the generated goal-models, and it proved to be correct.
- Transition T3: performing the goal-model structure check on the developed goal-model, it proved to be correct.
- Transition T4: performing the consistency (goal-conflict) checks, then either the goal-model is proved to be consistent or the user agrees.
- Transition T5: performing the completeness and reachability checks, then either the goal-model is proved to be complete and goal-reachable or the user agrees.
- Transition T6: Animating the goal-model and the user approves the animation.
- Transition T7: Performing Obstacle analysis.
- The light arrow transition stands for editing the goal-model and it automatically resets the goal-model state to the initial state.

6.4 Generating specifications (Phase III)

This phase is the final phase to automatically generate the output as a B specification. This automation hides the details of the B formal method and advanced logic and mathematics from the systems engineer. After he/she checks and tests the application goal-model, the tool can generate the

specifications. The tool provides two formal formats: the first is general invariants that consist of pre- and post-conditions of Boolean expressions built of the application's variable assignments and the actor (the agent) that performs such an invariant. The other format is B AMN machines. The tool implements an algorithm to transform the multi-level nature of goal-models to flat logic expressions as in B or other formal formats.

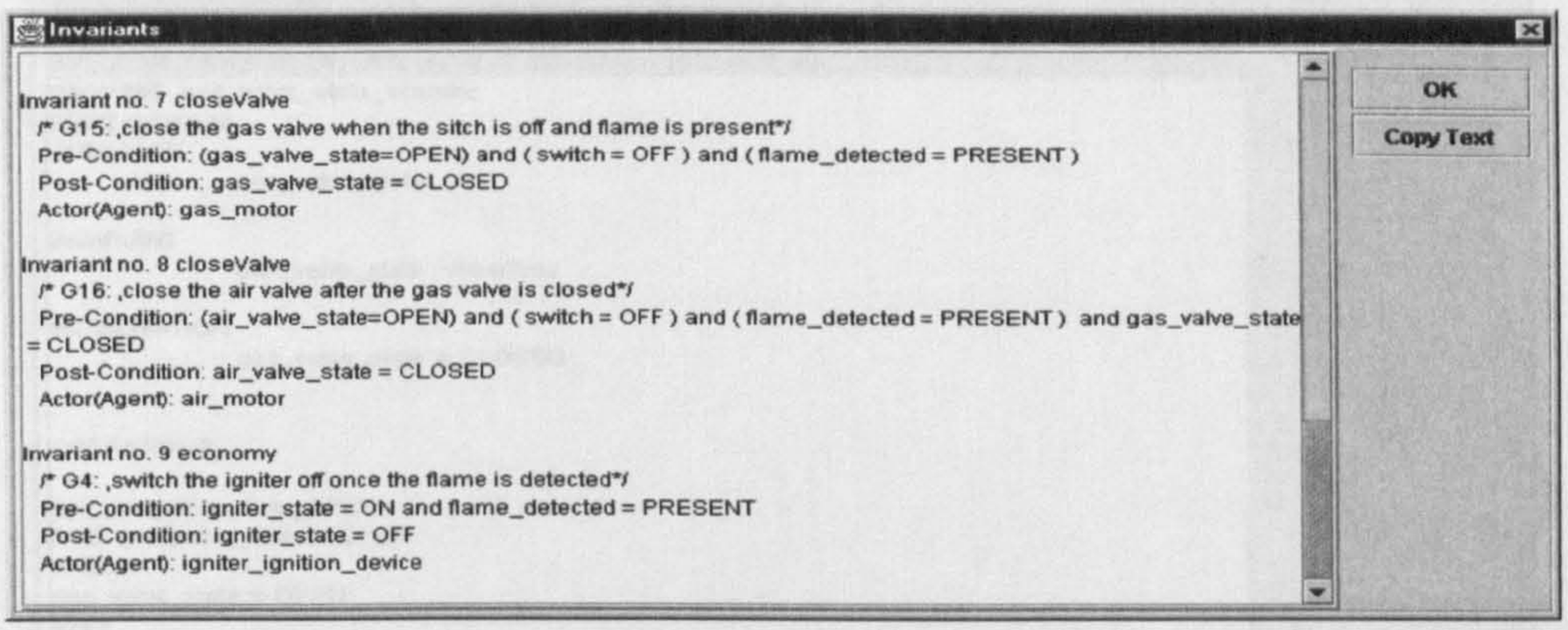


Figure 6.21, operation invariants

The translating algorithms use the definition of *postCond(G), preCond(G), accPreCond(G), accPostCond(G),* and *accCond(G)* functions described in chapter 5 and 5.43, 5.44, 5.45, 5.46, 5.47 and 5.48 rules to accumulate the pre-condition of each (functional and non-environmental) terminal goal from the relevant ancestor and predecessor goals. These terminal goals will consist of the operational invariants and will be the essential part of the *miancontroller* machine in the B specification.

6.4.1 Generating general operations/invariants (F15)

Invariants represent the logical input-output relationships between the application variables. The structure of the goal-model is not shown in the invariants. The user provides them as a formal specification form in the case that the user prefers to translate the requirements into another formal format. By selecting the Specification menu item/generate invariants, a dialogue box similar to the one in figure 6.21 will appear.

6.4.2 Generating B machines (F16)

As the final output of the GOPCSD tool, the tool provides formal specifications in the form of B AMN machines. By selecting the Specification/B machines sub-menu, a dialogue box similar to the one in figure A.40 will appear.

The tool generates the B machines according to the variables relationships within the goal-model. in the case that some output variable does not appear in the goal model, it will not be present in the specification as well the input variables. The tool generates the following machines and affix comments from the requirements information data:

- A machine involving the definition of the variables types; this information is collected from the variables' details created within the application or from the library.
- A set of actuator machines representing the actuators, based on the different operations that each agent could perform within the terminal goals.

- A main controller machine based on the goal-model hierarchy, representing the terminal goals as pre- and post conditions; the informal details of the terminal goals will be generated as comment lines at the proper sites within the B code to enable the software engineer to understand the operation of the application.

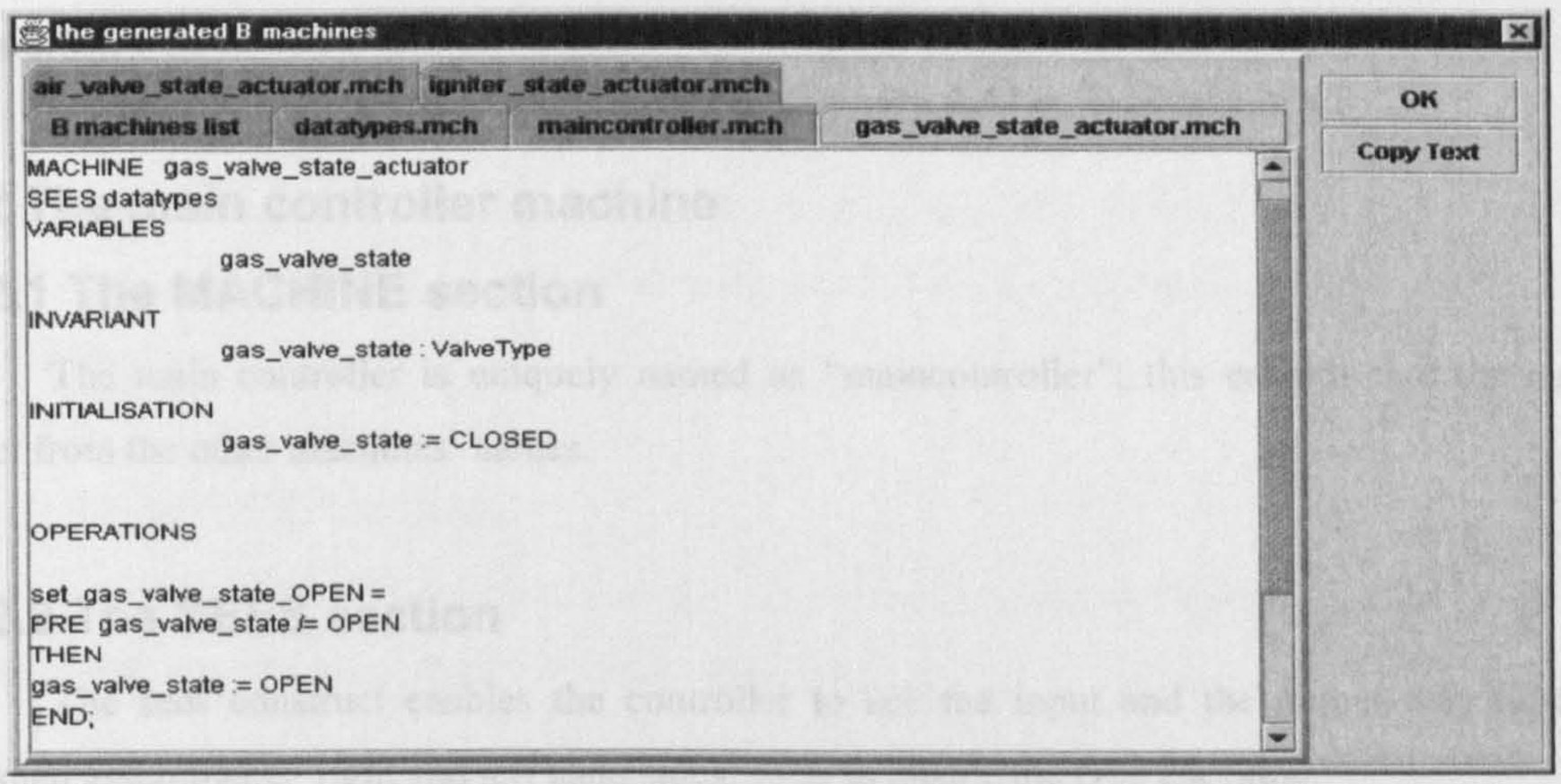


Figure 6.22, The Generated B machines

These B machines can be further refined and processed by the software engineer within the B toolkit environment with a high confidence that the systems engineer agrees with their requirements. As shown in figure 6.22, the tool provides a list of the generated B machine files inside the dialogue box's tabs; the user can browse the B files. In addition, the GOPCSD tool saves another version of these files on the directory of the requirements application.

6.5 Well-definedness of the B machines

This section explains that the generated B machines by the tool will obey the general constraints imposed by B in naming the B machines and operations. The translation from a goal-model into a B specification is carried out based on the variable, and data type lists and the goal-model. It depends on the formal definitions of the goal-model and the constraints that the GOPCSD tool enforces on the goal-model and the variables', values' and types' names. The tool generates three types of machines (datatype, actuator, main-controller); in the following sub-sections, we briefly explain each type.

6.5.1 The data type machine

6.5.1.1 The MACHINE section

The data type machine is uniquely named as "datatypes"; this ensures that its name will be distinct from the other actuator machines or the main controller machine.

6.5.1.2 The SETS section

This section translates the data types of the variables list within the goal-model to sets of enumerated data types. In the case that some of the output variables have the same data type, the data

type will appear only once. The tool ensures that the data types are uniquely defined and do not conflicted with the B AMN keywords. Each data type is defined as a set, which is composed of the individual enumerated values. In addition, these values are restricted by the tool to be valid Identifiers (begin with letters and do not include any special characters) and do not conflict with each other or the B keywords.

6.5.2 The main controller machine

6.5.2.1 The MACHINE section

The main controller is uniquely named as "maincontroller"; this ensures that the name is distinct from the other machines' names.

6.5.2.2 The SEES section

The sees construct enables the controller to see the input and the output data types and individual assignments' right and left hand sides, so as to enable the type checking.

6.5.2.3 The INCLUDES section

The INCLUDES section lists all the actuator machines that control the output variables, so as to enable the controller machine to invoke the appropriate operations of the actuators.

6.5.2.4 The OPERATIONS section

This section defines one operation; this operation has a pre-condition, which is always TRUE to ensure each cycle can be executed (continuity of reactive systems). It accesses the output variables in parallel. The assignment statements accessing or controlling the output variables are grouped by the output variable; each group has an "if then else" structure and runs in parallel with other groups.

This ensures that the same output variable will not be accessed from different sites (i.e. no more than one operation of the same machine (the actuator machines) will be simultaneously invoked (from the main controller machine as this is which is illegal in B).

In each of these parallel groups or sub-sections the structure is an "if then else" structure composed of the accumulated pre-conditions of the terminal goals accessing particular output variable and invokes one of the possible operations of the actuator machines' operations.

This structure ensures that only one action concerning an output variable will be taken at a time i.e. eliminating any chance of conflict. However, we assumed that the hard conflict cases are resolved within the second development phase supported by the GOPCSD tool, i.e. in the course of the conflict analysis.

Each parallel group invokes a SKIP command (i.e. being idle); this allows a space to insert a new specification (when refining this preliminary specification) when none of the nested pre-conditions is satisfied.

6.5.3 The actuator machines

Each actuator machine represents an output variable; it has operations to assign the different possible values the variable can have.

6.5.3.1 The MACHINE section

The actuator machines are named differently from each other because the name is generated from the distinct variable names (which is guaranteed by the GOPCSD tool). The name is different from the "datatypes" and "maincontroller" as well since it has the word "actuator" as a suffix, after the output variable name.

6.5.3.2 The SEES section

This is to enable the machine to see the variable data type; and makes it possible to prove the output variable always assign values from the same data type.

6.5.3.3 The VARIABLE section

This defines the variable name, which is guaranteed by the GOPCSD to be unique; no two variables can have the same name and none can conflict with any Keyword of the B language; this is hardwired within the tool when creating or modifying a variable name.

6.5.3.4 The INVARIANT section

This is to guarantee the variable always has values from its data type; since each variable will be accessed only by the actuator machine and the main controller was restricted to invoke the actuator operations to change the value of the variables, this proves that the variables will always have values within their data types.

6.5.3.5 The INITIALISATION section

This information is translated from the variable data (initial value); since it is achieved by selecting one of the possible values, the output variables are initialised satisfying the invariants of their corresponding machines.

6.5.3.6 The OPERATIONS section

Each operation of the actuator machine is intended to assigning one possible value to the variable. Thus, these operations always respect the invariant of the machine.

6.5.4 Goal-Model and B machines, situation by situation

In the following table we list situations in the goal-model and the corresponding ones of the generated B machines.

Table 6.1, goal model and B machines situation by situation

Situation	The goal-model	The B Machines
Initialisation	Each variable is assigned its initial value	In the actuator machines, each variable is assigned its initial value
One goal is active	When one goal is activates it assigns an output variable a value	The main controller machine will invoke the appropriate operation from the actuator to assign the values
More than one goal is activated	Each of the activated consistent goals assigns a value to different output variables	Two (or more) conditions are satisfied within the main controller operation in two or more different parallel groups and the different actuator operations are invoked
Idle	None of the goals of the goal-model is activated	In the main controller machine, each of the parallel sections of the main operation will invoke *SKIP* (i.e. being idle)
Soft goal-conflict	When two goals accessing the same output variable assign the same value	Within the nested if then else structure, only one of these goals which appears before the other(s), will invoke the same operation
Hard conflict	The goal-model should not be accepted since its behaviour is not consistent	Among the conflicting goals, only one of them will be allowed to invoke an actuator operation.

6.6 Conclusions

The GOPCSD tool implements the requirements development phases. The tool can be considered as an integrated environment where the systems engineer can be guided to structure, test, and validate the process control application's requirements, thus preparing the stage for the software engineer to continue the development from the software perspective. The tool attempts to increase the feedback provided to the user at the different stages of the application development to enable him/her to generate B specification machines corresponding to complete, consistent, valid and user-agreed requirements.

Case Study I
The Gas Burner System

7

In this chapter, we present the first case study to examine both the GOPCSD method and its supporting tool. The case study examines the flexibility of the GOPCSD tool to enable the user to develop different versions of the control program. It shows also how the feedback guidance can enable the user to modify the requirements easily, and effectively. The validation enables an early observation of the application's normal behaviour as well as behaviour under faults that can happen to its components during operation.

7.1 Introduction

The first case study is a general-purpose gas burner system [Lano and Sanchez 97]. It combines basic input/output relations with safety and power-minimisation issues. The system is composed as shown in figure 7.1, from a gas valve, an air valve, a switch, an igniter, and a flame detector. The switch is used to start and stop the operation of the burner. The igniter is used to produce a spark, while the flame detector is used to sense the existence of the flame. The flame is kept burning by providing a continuous stream of air and gas.

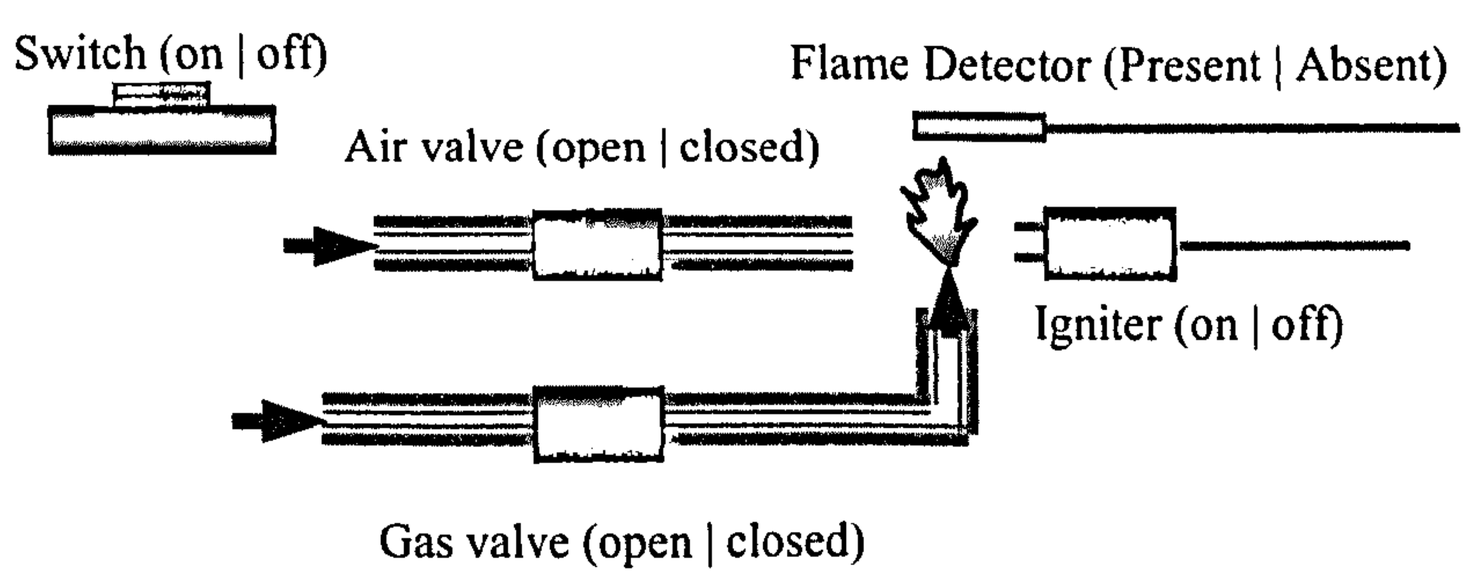


Figure 7.1, The Gas burner system

As described earlier in chapter 4, the gas burner functions can be extracted from the system documentation; or can be provided in natural language form by the systems engineer. Although the tool can guide its user to elicit a complete set of goals, the user can target the following apparent functions: starting up when required, shutting down when required, maintaining safety conditions, maximising the lifetime of the igniter, and keeping the flame burning during operation.

These functions will appear within the imported templates where they are combined to form higher-level goals addressing the different aspects of the entire gas burner system. Each of them will be expressed as a high-level goal that should be refined to a level where the imported components requirements are integrated to serve as lower-level goals. In the case that the imported templates do not contain any of these functions, the user should define it in a new goal-model. Afterwards, the tool should guide the user to produce a single complete goal-model specifying the entire gas burner system.

7.2 Constructing the goal-model

In the GOPCSD tool, the development of the gas burner application starts by creating a new application. This will clear the application component, variable, agent and goal-model lists. Then, the user can start importing the related components and templates, defining the rest of the application functions, and refining and combining the different goals until reaching a state where the user constructs a single complete goal-model that specifies the entire application and fulfils the various requirements aspects.

7.2.1 Importing the components

The first step in constructing the application requirements is to identify the components that constitute the application. For this purpose, the tool enables the user to browse the existing libraries and investigate their contents. As shown in figure 7.1, two valves: air and gas, an igniter to produce the spark, and two sensors: flame detector and switch can be identified.

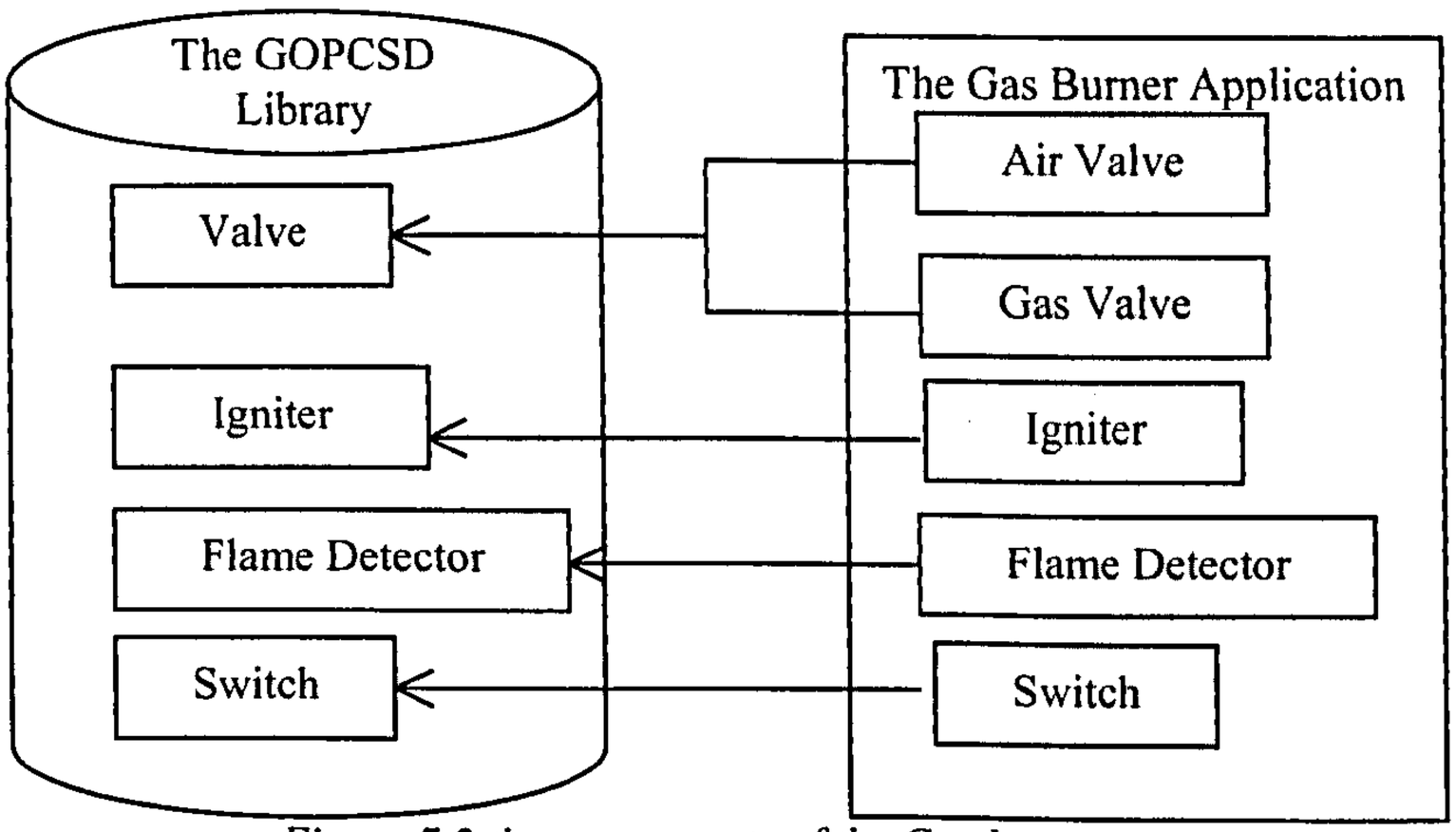


Figure 7.2,the components of the Gas burner system

Although the two valves of the gas burner system are not physically identical, they still possess the same low-level operations. Thus, the two valves can be abstractly represented by two

instances of a single library component valve as shown in figure 7.2, where we indicate the components to be imported from the library. This abstraction in modelling cannot be applied if the physical differences of components imply different operational constraints.

Because the components do not share any variables or agents, the user can add them separately and he/she does not have to map any of their details. As shown in figure 7.3, when importing each component, the user can type the component name in a dialogue box like as shown in figure 7.3, without checking the checkbox to add component details of the component as they appear in the library, instead of deciding to add/add and rename/map each variable and agent separately. This shortens the import time.

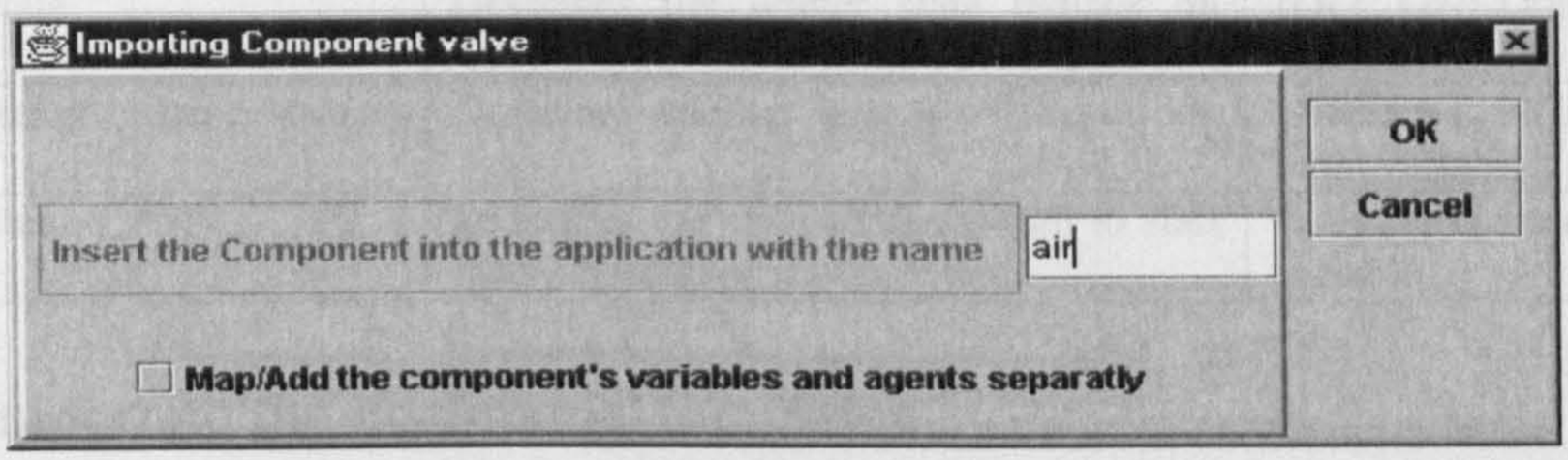


Figure 7.3, Importing the valve component and changing its name

Thus, after the user imports the valve component twice for the air and gas valves, he/she similarly imports the other components ignitier, flame_detector and switch. The last two components may be imported or not, as each of them contains only a single input variable and neither agents nor goal-models; however, it is recommended to add them rather than defining isolated variables to have a better understanding of the composition of the application. At this level, the application contains the requirements elements, as shown in figure 7.4, which shows the details of the gas burner application after importing the components from the library. As shown in the figure, the application contains five components, three agents (one to control each valve and one for the ignition device), five variables (three output variables (one for each valve and one for the ignition) and two input variables (one for the switch and one for the flame detector) and, finally, six low-level goal-models (two for each valve and two for the ignition device). The details of component goal-models are not shown; however, the user can list the variables and agents, grouped by components. Or, alternatively, the user uses the "show the details" of each component menu item to list the related agents, variables and goal-models.

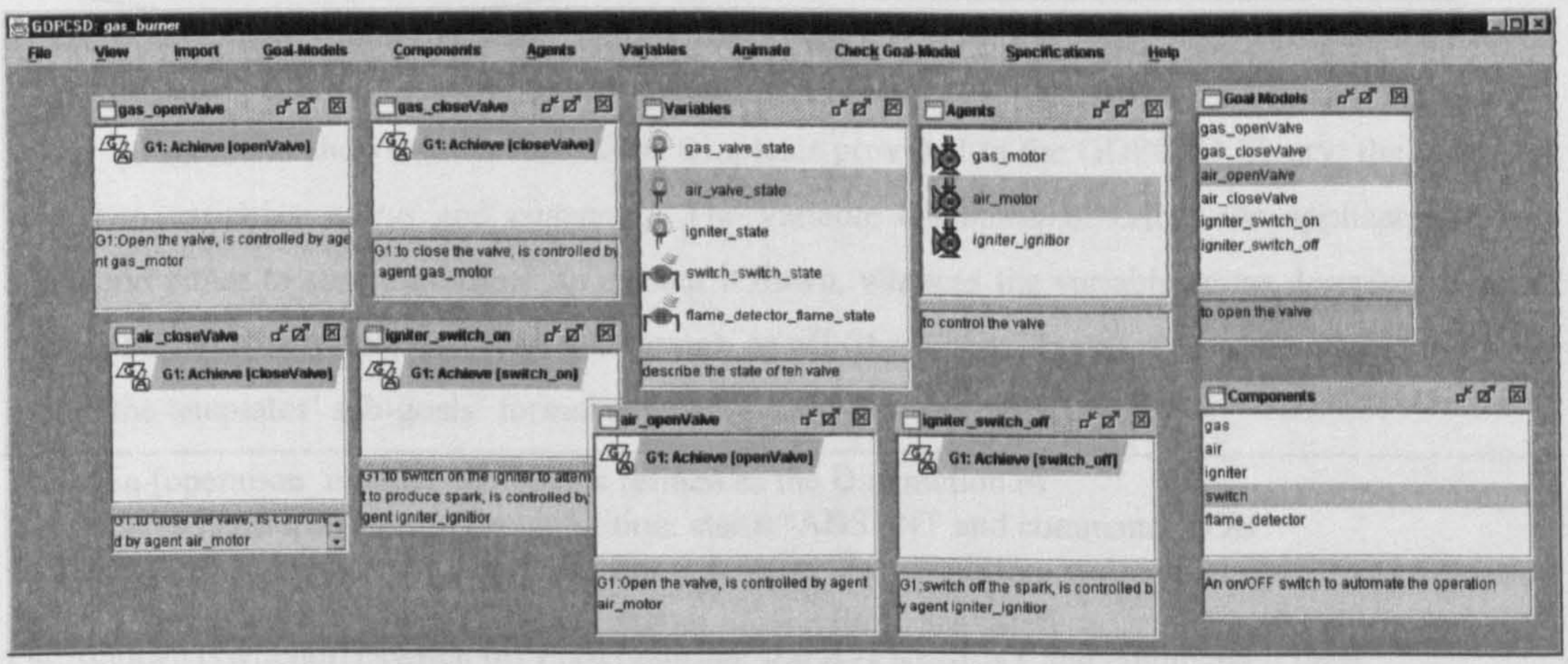


Figure 7.4, the GOPCSD desktop after importing the components

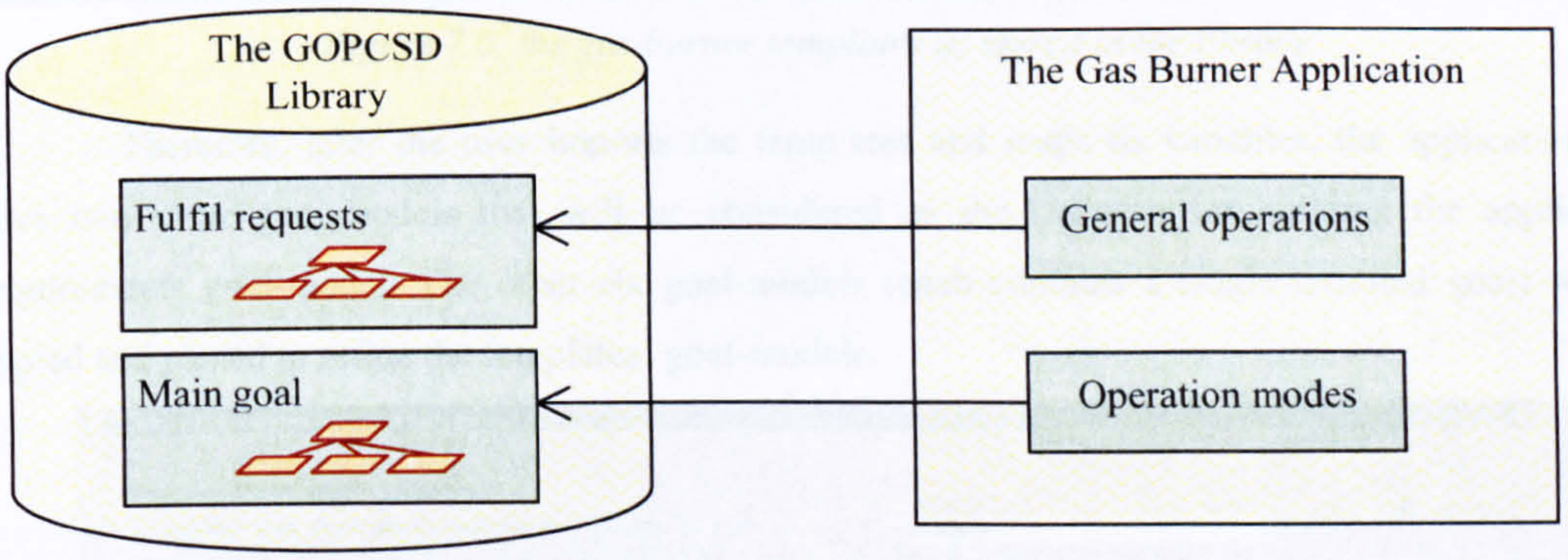
In table 7.1, the application variables gathered from the different components are listed in detail. These variables are essential for formalising the requirements and enabling the different validation and consistency checks.

Table 7.1, related variables for the gas burner system.

Name	Description	Type
air_valve_state	Describes the state of the air valve	{CLOSED, OPEN} Output variable
gas_valve_state	Describes the state of the gas valve	{CLOSED, OPEN} Output variable
igniter_state	Describes the igniter state (equals on when attempting to produce the spark)	{OFF, ON} Output variable
flame_detector_flame_state	Describes whether there is a flame or not	{ABSENT, PRESENT} Input variable
switch_state	Describes the state of the switch to start the system up or shut it down.	{OFF, ON} Input variable

7.2.2 Importing the goal-model templates

The user is advised not only to import components, but also to import goal-model templates from the GOPCSD library. These library templates guide the construction of the goal-model. In the gas burner system, the library provides two templates as shown in figure 7.5. One template describes the general aspects of the application like safety, operational and economic aspects. The other template describes the fulfilment of the user commands. The tool provides a chance to understand the abstract functions and their general description before importing them.



7.5, importing the templates from the library

Figure 7.6 shows the details of the templates provided in the GOPCSD library; the templates have two variables: *status* and *command*. The variable *command* describes the application user's command either to start the burner up or shut it down, whereas the variable *status* describes whether the gas burner is currently on (there is a flame), or off (there is no flame). These two variables appear within the templates' sub-goals' formal details, as follows:

Maintain [operation_modes]: the goal is refined as the Disjunction of
G2:Achieve [startup]:start up, Pre-condition: status=ABSENT and command = ON
G3:Maintain [flame_burning]:keep the flame burning Pre-condition: status=PRESNET and command = ON
G4:Achieve [switchoff]:switch off Pre-condition: status=PRESENT and command = OFF

We note that neither of the two templates have agents since they contain only non-terminal goals. Although the sub-goals G2, G3 and G4 in each template may appear as terminal goals since they do not have sub-goals, they will be refined within the application, when the user decides on the components of the application.

In addition, we can notice that the application variables *switch_state* and *flame_detector_flame_state* perform exactly the same functions as the templates' two variables. Therefore, one should not append the templates' variables to the application variable list, but, instead, map the application variables to the template variables. Thus, the variables *switch_state* and *flame_detector_flame_state* should replace the *command* and *status,* respectively, in the template's goals' pre-conditions. The GOPCSD tool enables its user to map/add the template variables and agents as shown in figure 7.7, where a dialogue box for mapping variable *status is shown;* the user should select the variable *flame_detector_flame_state* to map to. Similarly, the application variable *switch_state* should map to the template variable *command*.

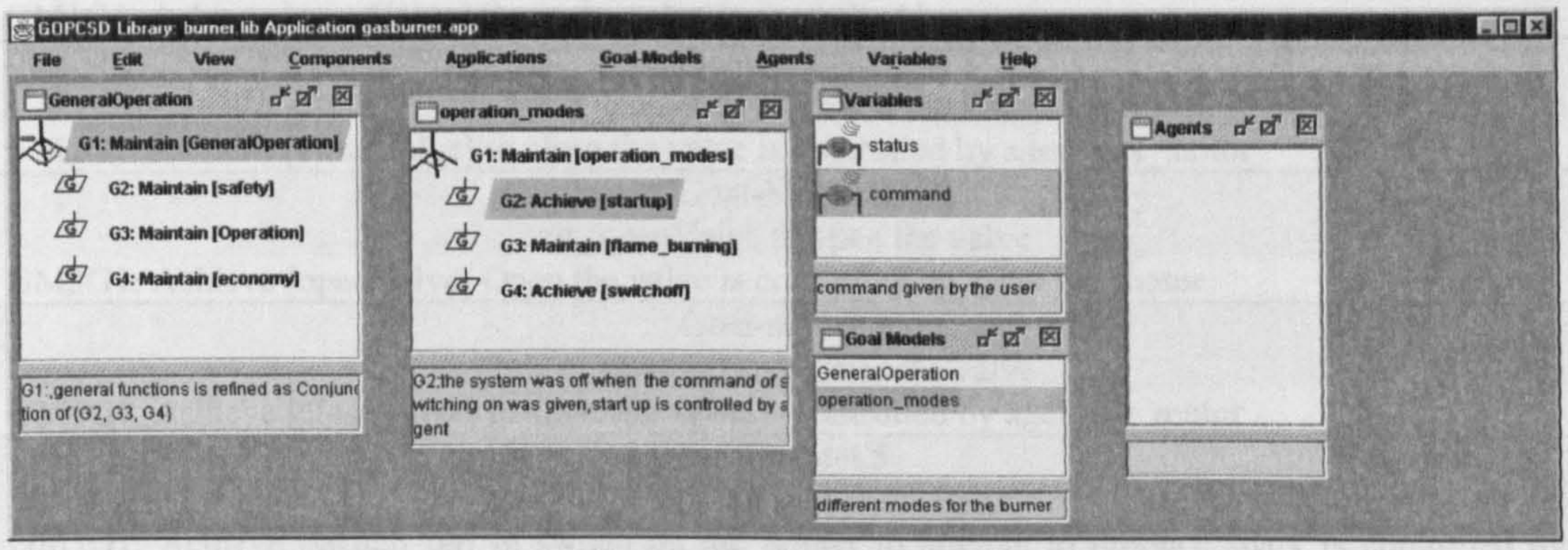


Figure 7.6, the gas burner templates as stored in the library

Therefore, after the user imports the templates and maps its variables, the application will have two more goal-models that will be considered as the skeleton for building the application requirements goal-model. The other six goal-models (each contains a single terminal goal) will be copied and pasted to refine the templates' goal-models.

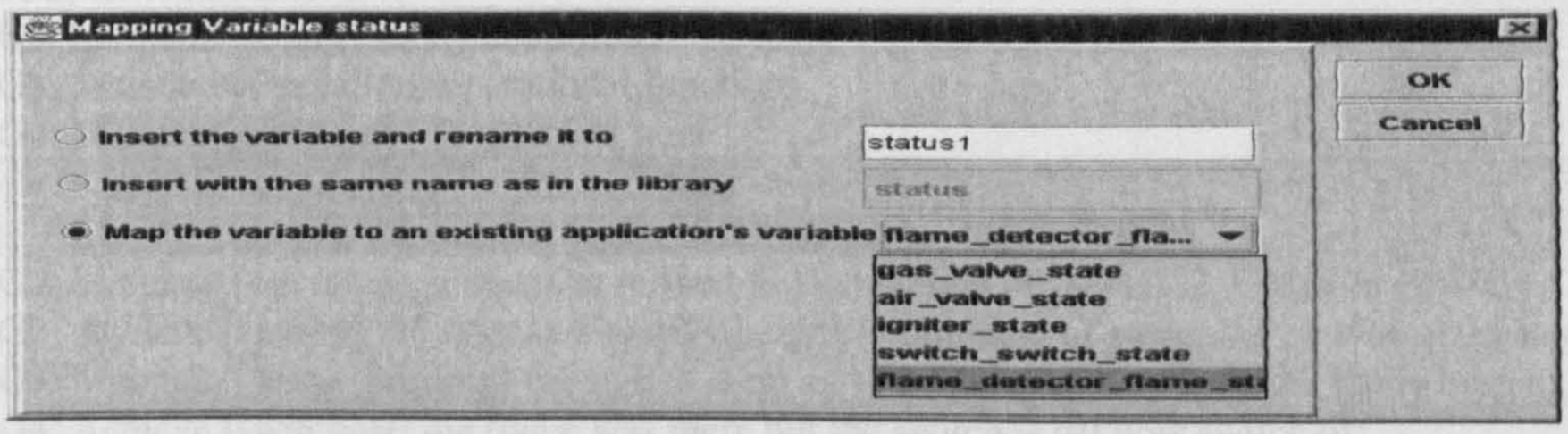


Figure 7.7, mapping the template variable status

7.2.3 Incorporating the application's goals

After the user imports the relevant components and templates, he/she should compare the application functions, as indicated in section 7.1, to the existing application's goals contained within the different goal-models. The user can identify the following five goals (functions) from the system documentation:

Table 7.2, the extracted goals (functions) from the system documentation

Goal/function name	Goal Function
G_a	Start up when requested
G_b	Keep the flame burning
G_c	Shut down when requested
G_d	Maximize the lifetime of the igniter
G_e	Maintain the safety conditions

The application goals are listed in table 7.3 where the goals are grouped according to the containing goal-models and indexed in a breadth-first manner using the container goal-model index as prefix. The imported templates are added to the application and have identifiers such as goal-model 7 and goal-model 8.

Table 7.3 the application goals after importing templates and components

Goal model 1 gas_openValve to open the valve
GM1G1: Achieve [openValve] Open the valve is controlled by agent gas_motor
Goal model 2 gas_closeValve to close the valve
GM2G1: Achieve [closeValve] to close the valve is controlled by agent gas_motor
Goal-Model 3 air_openValve to open the valve
GM3G1: Achieve [openValve] Open the valve is controlled by agent air_motor
Goal-model 4 air_closeValve to close the valve
GM4G1: Achieve [closeValve] to close the valve is controlled by agent air_motor
Goal model 5 igniter_switch_on switch the igniter on
GM5G1: Achieve [switch_on] to switch on the igniter to attempt to produce spark is controlled by agent igniter_ignitior
Goal-model 6 igniter_switch_off to switch off the igniter after producing the spark
M6G1: Achieve [switch_off] switch off the spark is controlled by agent igniter_ignitior
Goal-model 7 GeneralOperation descripe generally the high level functions of the burner
GM7G1: Maintain [GeneralOperation] general functions; this goal is refined as Conjunction of {GM7G2, GM7G3, GM7G4} GM7G2: Maintain [safety] safety conditions GM7G3: Maintain [Operation] operational functions GM7G4: Maintain [economy] economical goal
Goal-model 8 operation_modes different modes for the burner
GM8G1: Maintain [operation_modes] is refined as Disjunction of {GM8G2, GM8G3, GM8G4} GM8G2: Achieve [startup] the system was off when the command of switching on was given start up GM8G3: Maintain [flame_burning] the switch is on and the flame is there keep the flame burning GM8G4: Achieve [switchoff] the flame was there but the switch is off now switch off

It is important for the user to map between the required goals/functions G_a, G_b, G_c, G_d and G_e and the application existing goals listed in table 7.2. A little consideration can show that goal G_a can be represented by goal GM8G2, G_b can be represented by goal GM8G3, G_c can be represented by goal GM8G4, and G_e can be represented by goal GM7G2. On the other hand, goal G_d clearly cannot be represented in the current situation by any of the listed goals. The development can proceed now by accessing these application goals and editing/completing their informal and formal definitions, if

possible, through specifying the pre- and post-condition of each goal. The user can create a new workspace for goal G_d and name it maximize igniter lifetime as shown in figure 7.8. The figure shows the new goal-model dialog to the left and the newly created goal-model on the desktop of the GOPCSD tool. Having created this goal-model, goal G_d is now represented by GM9G1, the root goal of the newly created goal-model. Later, all these goal-models have to be combined to form a complete goal-model that specifies the entire gas burner system. Each of these identified goals has to be classified as either maintain, achieve, cease, or avoid. The tool provides templates for these four goals types based on Temporal Logic [Mellor and Ward 85].

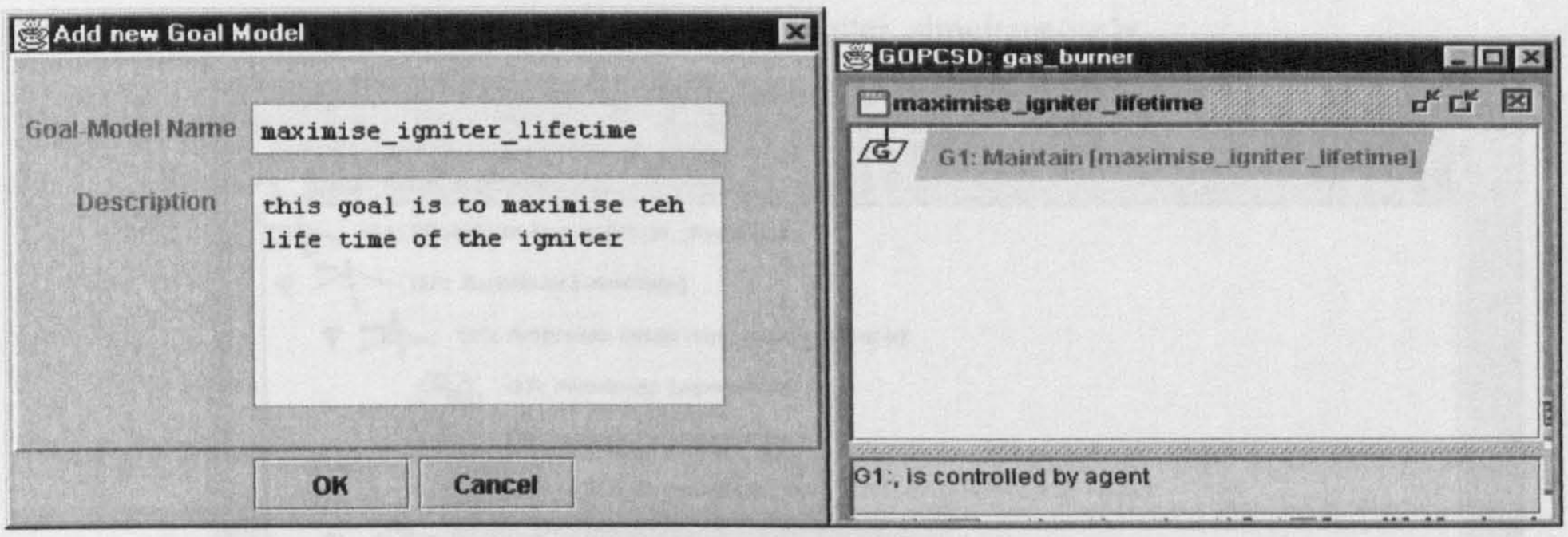


Figure 7.8, creating a new goal- model for maximizing the lifetime of the igniter

7.2.4 Refining the goal-model

The next step is to complete the goal-models by refining the branch goals into sub-goals with narrower scope using the appropriate refinement patterns. This refinement process should stop when each terminal has a scope narrow enough to be controlled directly by a single agent. The application contains nine goal-models: goal-models 1, 2, 3, 4, 5 and 6 are associated with of the components while goal-models 7, 8, and 9 constitute the skeleton on which the user is going to build the application requirements. The refinement step involves refining the latter three goal-models.

7.2.4.1 Operation (goal-model 8) refinement

In many cases a situation can arise when the user prefers to try different solutions. The tool provides the alternative refinement pattern for this purpose. It enables the user to compact the different solutions into a single but compound goal-model to reduce the effort required to duplicate the common sub-goal-models between the different solutions.

For example, goal GM8G2, starting the system up, can be accomplished in two different ways: either sequentially opening the air valve, the gas valve, and then switching on the igniter or alternatively, open the two valves and switch the igniter on, simultaneously. The sequential alternative has the advantage of controlling the transitional states the system can reach until it settles, while the simultaneous alternative has the advantage of shortening the starting up and shutting down time. This can be achieved by selecting the goal GM8G2 within goal-model 8 and changing its refinement pattern to be alternative, then, adding two new sub-goals to goal GM8G2.

The newly generated goals will have the indices GM8G5 and GM8G6. Goal GM8G2, that represents the starting up of the burner, can be specified as an achieving goal that gets activated when the user switches the system on while there is no flame detected; since goal GM8G5 has to control the output variables *air_valve_state*, *gas_valve_state*, and *ignitor_igniter_state*, it needs further refinement, until each sub-goal controls a single output variable or a group of variables that are controlled by a single agent. Thus, the sequential alternative goal GM8G5 can be refined into three sub-goals that open the air valve, open the gas valve, and switch on the igniter, sequentially. Alternatively, the alternative simultaneous goal GM8G6 can be refined into three sub-goals that open the air valve, open the gas valve, and switch on the igniter, simultaneously.

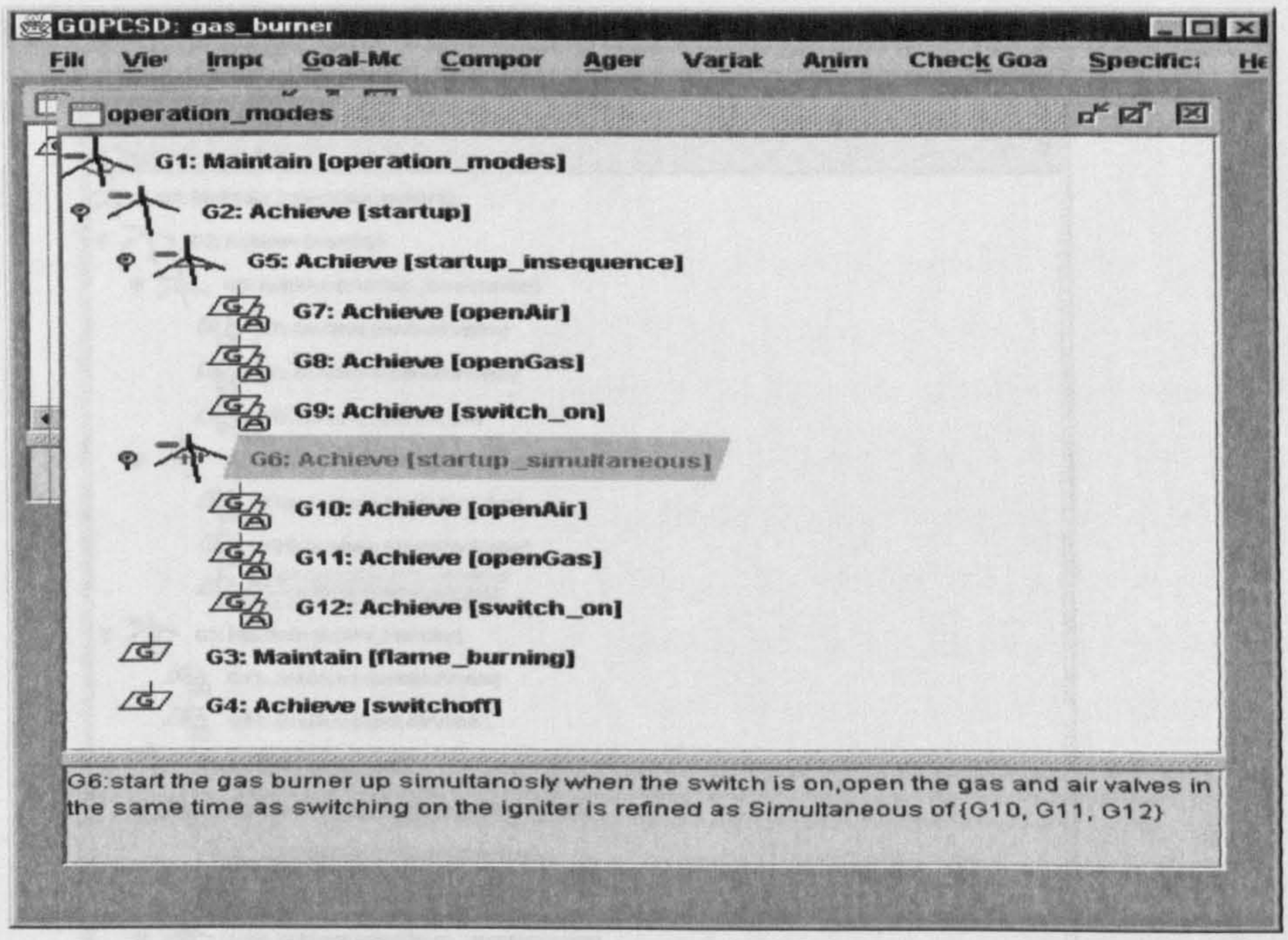


Figure 7.9, the operation modes goal-model after refining goal GM8G2

The user should change the refinement pattern of goal GM8G5 to the sequence pattern and that of goal GM8G5 to the simultaneous pattern. It is important to notice that the sub-goals of the sequential or the simultaneous alternatives are the gas valve's, air valve's and igniter's terminal goals. Thus, the user can reduce considerable effort and shorten the development time by copying the terminal goals GM3G1 (open the air valve), GM1G1 (open the gas valve) and GM5G1 (switch the igniter on) and pasting them as sub-goals under goals GM8G5 and GM8G6. After this refinement process, goal GM8G2 can be considered as a fully refined goal, since all of its descendant terminal goals are assigned to agents. Figure 7.9 shows goal-model 8 after completing this refinement process.

Goal GM8G4, meant to keep the flame burning under the appropriate conditions, can be formulated as a maintaining goal that controls the gas and air valves; if by any chance one of these two valves is closed while the switch is still on and there is an existing flame, this goal would attempt to open the valve again. Because the two valves are controlled by different agents, the goal GM8G4 needs further refinement to two goals: goal GM8G7 as a copy of goal GM3G1 (open the air valve) and GM8G8 as a copy of GM1G1 (open the gas valve); each of them controls one of the valves and attempts to keep it open.

Goal GM8G4 represents shutting the gas burner down; it is activated when the user switches the system off while there is a flame detected; similar to goal GM8G2, goal GM8G4 for switching off has an achieving type and can be refined to either sequential or simultaneous alternatives; the sequential alternative goal GM8G9 can be refined to two sub-goals: the first sub-goal should close the gas valve as a copy of goal GM2G1 (close the gas valve), while the second sub-goal should close the air valve as a copy of goal GM4G1 (close the air valve); thus, the flame should go off.

Alternatively, within the simultaneous alternative goal GM8G10, the two valves can be closed simultaneously in two different goals. Table 7.4 lists the goals of goal-model 8 (operation modes), after refining goals GM8G2 (start up when requested), GM8G3 (keep the flame burning), and GM9G4 (shut down when requested).

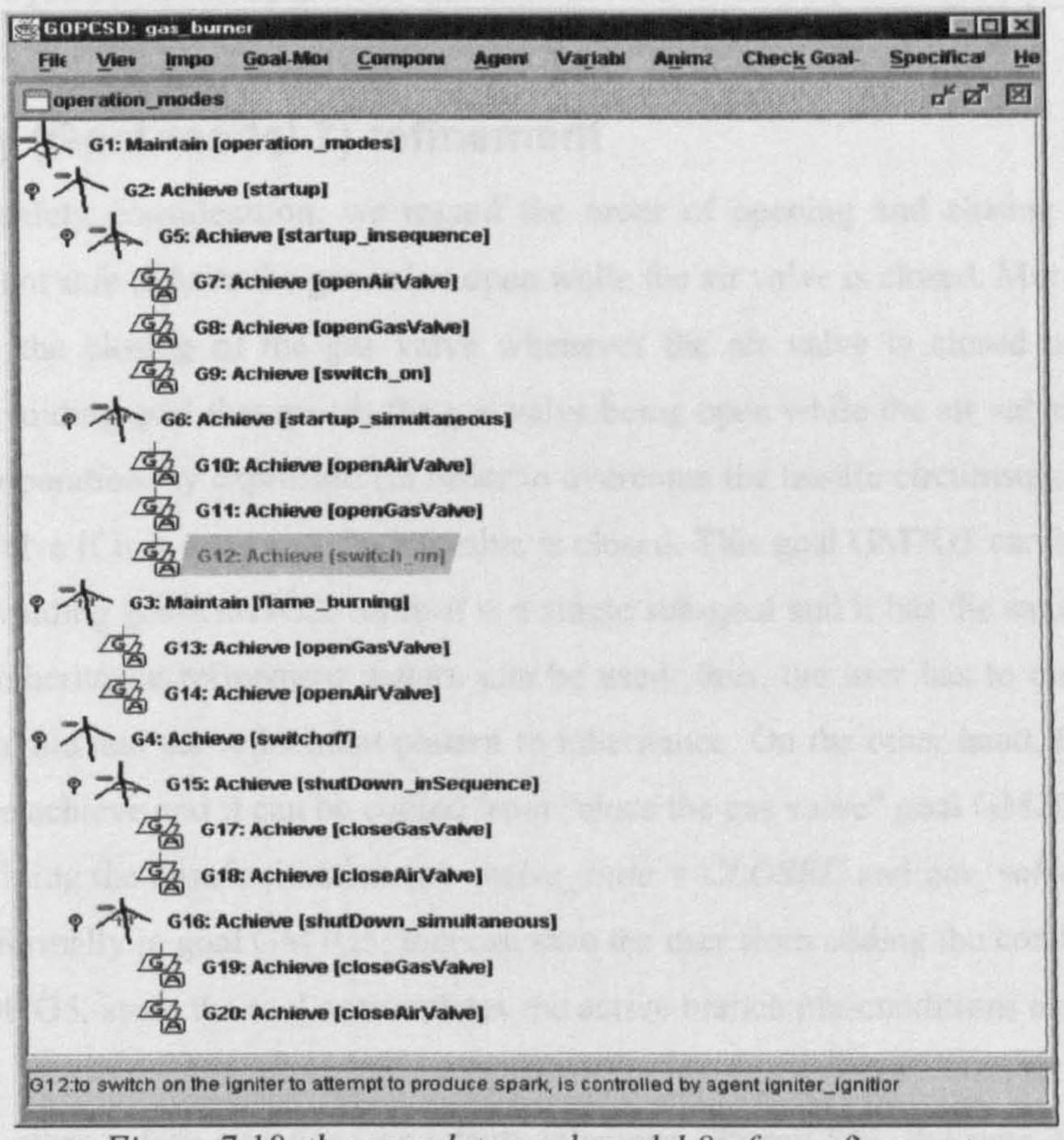


Figure 7.10, the complete goal-model 8 after refinement

Table 7.4, the goals of the operation-modes after refinement

8- operation_modes :different modes for the burner
GM8G1: Maintain [operation_modes] is refined as Disjunction of {GM8G2, GM8G3, GM8G4} GM8G2: Achieve [startup] the system was off when the command of switching on was given start up is refined as Alternative of {GM8G5, GM8G6} GM8G5: Achieve [startup_insequence] start up in sequence open the air valve, then the gas valve the switch on the igniter is refined as Sequence of {GM8G7, GM8G8, GM8G9} GM8G7: Achieve [openAirValve] Open the valve is controlled by agent air_motor GM8G8: Achieve [openGasValve] Open the valve is controlled by agent gas_motor GM8G9: Achieve [switch_on] to switch on the igniter to attempt to produce spark is controlled by agent igniter_ignitior GM8G6: Achieve [startup_simultaneous] start the gas burner up simultaneously when the switch is on open the gas and air valves in the same time as switching on the igniter is refined as Simultaneous of {GM8G10, GM8G11, GM8G12} GM8G10: Achieve [openAirValve] Open the valve is controlled by agent air_motor GM8G11: Achieve [openGasValve] Open the valve is controlled by agent gas_motor GM8G12: Achieve [switch_on] to switch on the igniter to attempt to produce spark is controlled by

agent igniter_ignitior
GM8G3: Maintain [flame_burning] the switch is on and the flame is there keep the flame burning is refined as Simultaneous of {GM8G13, GM8G14}
GM8G13: Achieve [openGasValve] Open the valve is controlled by agent gas_motor
GM8G14: Achieve [openAirValve] Open the valve is controlled by agent air_motor
GM8G4: Achieve [switchoff] the flame was there but the switch is off nowswitch off is refined as Alternative of {GM8G15, GM8G16}
GM8G15: Achieve [shutDown_inSequence] when the switch is off and there is a flame detected shut the gas and the air valves is refined as Sequence of {GM8G17, GM8G18}
GM8G17: Achieve [closeGasValve] to close the valve is controlled by agent gas_motor
GM8G18: Achieve [closeAirValve] to close the valve is controlled by agent air_motor
GM8G16: Achieve [shutDown_simultaneous] when there is a flame and the switch is off shut down gas and air valves is refined as Simultaneous of {GM8G19, GM8G20}
GM8G19: Achieve [closeGasValve] to close the valve is controlled by agent gas_motor
GM8G20: Achieve [closeAirValve] to close the valve is controlled by agent air_motor

7.2.4.2 Safety (Goal-model 7) refinement

For the safety consideration, we regard the order of opening and closing the gas and air valves, since it is not safe to have the gas valve open while the air valve is closed. Moreover, a separate goal that ensures the closing of the gas valve whenever the air valve is closed can be expressed separately as an avoiding goal that avoids the gas valve being open while the air valve is being closed. This goal can be operationally expressed (in order to overcome the unsafe circumstance, if it happens) as close the gas valve if it is open and the air valve is closed. This goal GM7G5 can be expressed as a sub-goal of the avoiding goal GM7G2. Since it is a single sub-goal and it has the same function as the parent goal, the inheritance refinement pattern can be used; thus, the user has to change the type of goal GM7G2 to avoid and the refinement pattern to inheritance. On the other hand, for goal GM7G5, the type should be achieve and it can be copied from "close the gas valve" goal GM2G1. However, the pre-condition defining the unsafe situation (*air_valve_state* = *CLOSED* and *gas_valve_state* = *OPEN*) should be stated formally in goal GM7G5; this can save the user from adding the condition again to the terminal goal GM7G5, since the tool accumulates the active branch pre-conditions as explained earlier in chapter 5.

For the other safety situations and fault tolerance [Storey 96], the user should be able to express the safety ensuring paradigms as goal-models or segments of goal-models that can be combined with the operation segments, as explained in the following sections.

7.2.4.3 Lifetime of the igniter (goal-model 9) refinement

With respect to goal GM9G1, increasing the igniter lifetime can be realised by reducing the number of attempts it makes to produce the spark.

And, apparently, after the flame is produced, not attempting to produce sparks. This can result in formulating goal GM9G1 as an avoiding goal that avoids switching the igniter on if a flame is present, when the burner is starting up. Again this goal can be refined to an operational goal GM9G2 that switches off the igniter as soon as a flame is detected. This can guide the user to choose the pre-condition of goal GM9G2 to be (*switch* = ON and *flame_detector_flame_state* = PRESENET and *igniter_state* = ON) and its action or post-condition to be *igniter_state* = OFF.

Thus, goal GM9G2 can be produced as a copy of the switching of igniter's terminal goal GM6G1. Figure 7.11 shows goal-models 7 and 9 after adding the two terminal goals GM7G5 and GM9G2. Table 7.5 lists the goals of these two goal-models.

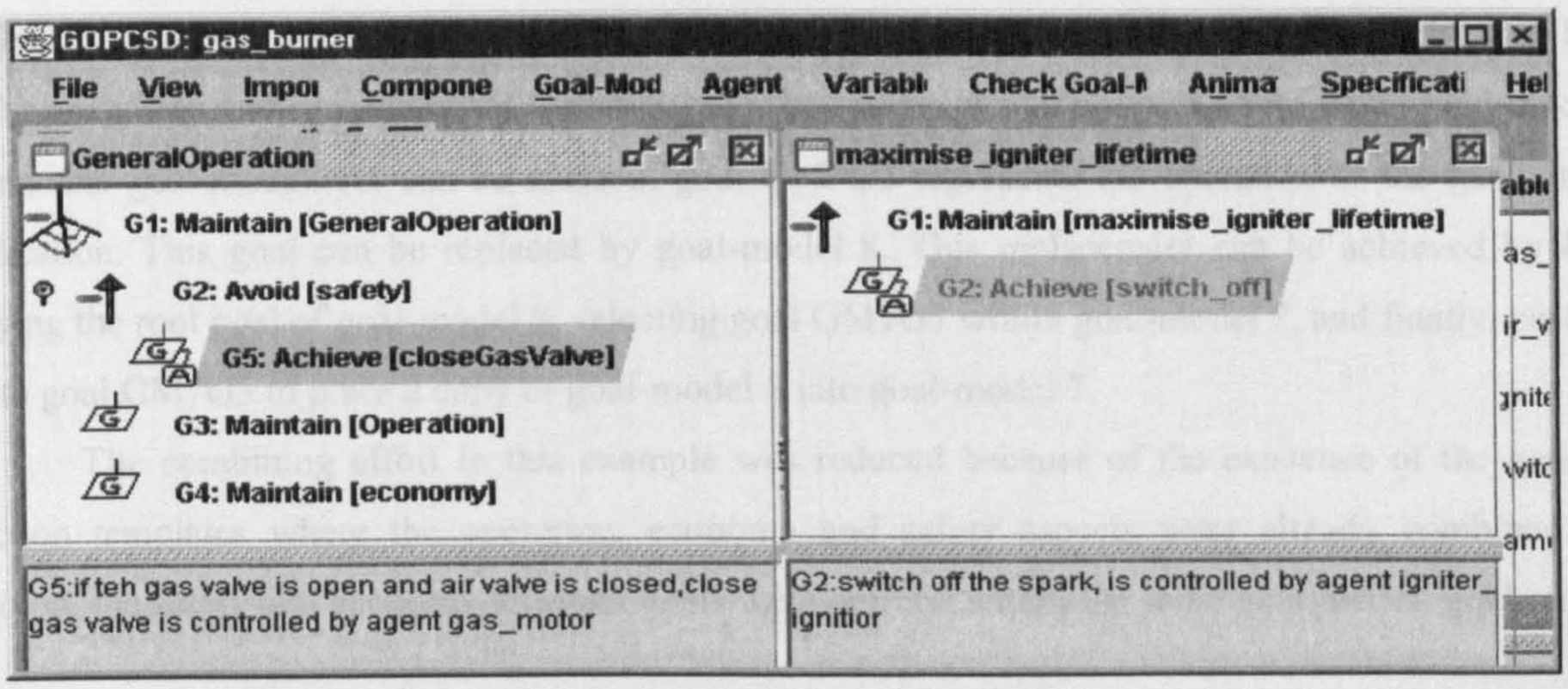


Figure 7.11, safety goal-model to the left and igniter lifetime goal-model to the right

Table 7.5, the goals of goal-models 7and 9

Goal-model 7 GeneralOperation :descripe generally the high level functions of the burner
GM7G1: Maintain [GeneralOperation] general functions is refined as Conjunction of {GM7G2, GM7G3, GM7G4} GM7G2: Avoid [safety] avoid the gas valve open when the air valve is closed is refined as Inheritance of {GM7G5} GM7G5: Achieve [closeGasValve] if the gas valve is open and air valve is closed, close gas valve is controlled by agent gas_motor GM7G3: Maintain [Operation] operational functions GM7G4: Maintain [economy] GM7G6: Achieve [switch_off] switch off the spark is controlled by agent igniter_ignitior
Goal-model 9 maximise_igniter_lifetime
GM9G1: Maintain [maximise_igniter_lifetime] if the flame exists and switch is ON, economical goal is refined as Inheritance of {GM9G2} GM9G2: Achieve [switch_off] switch off the spark is controlled by agent igniter_ignitior

7.2.5 Combining goals/goal-models

After the tool guided the user to refine the various goal-models, the next step is to combine them into a single goal-model that specifies the entire application.

The third sub-goal GM7G4 of the general functions goal-model imported from the library points to an important aspect, which is economical operation; the economy issue covers a wide area that includes the selection of the valves models, the mixture of the gas and air and other issues; however, at this level the user would be more interested in the running cost of the application like the lifetime issues of the different components and the power consumption. This guides the user to correlate the two goals GM7G4 (economy) and GM9G1 (maximise the lifetime of the igniter); goal GM9G1 contributes to goal GM7G4; therefore, it can be combined as a sub-goal. If there are no other contributing goals to the economy issue, the refinement pattern can be inheritance. Thus the two goal-

models 7 and 9 can be combined in this way, by copying the root goal of goal-model 9, then selecting the economy goal GM7G4 and pasting it as a new child.

The root goal of goal-model 7, the general functions of the gas burner, can be regarded as the top-level goal of the gas burner system because it addresses the different aspects of the application like operation, safety and running cost. Therefore, the user needs to place copies of the other goal-models within this goal-model. As can be noticed, goal GM7G3 represents the operation of the gas burner application. This goal can be replaced by goal-model 8. This replacement can be achieved by first copying the root goal of goal-model 8, selecting goal GM7G3 within goal-model 7, and finally, pasting it into goal GM7G3 to place a copy of goal-model 8 into goal-model 7.

The combining effort in this example was reduced because of the existence of the general function templates where the operation, economy and safety aspects were already combined. In addition, the safety and economy terminal goals were defined within the same goal-model, goal-model 7.

If this template was not available in the library, the user would define the safety and economy goals separately in newly created goal-models, then he/she would create a new goal-model and copying in turn each goal-model's root-goal and pasting it as a new child for the root-goal of the new goal-model. Figure 7.12 shows the goal-model after combining goal-model 9 and goal-model 8.

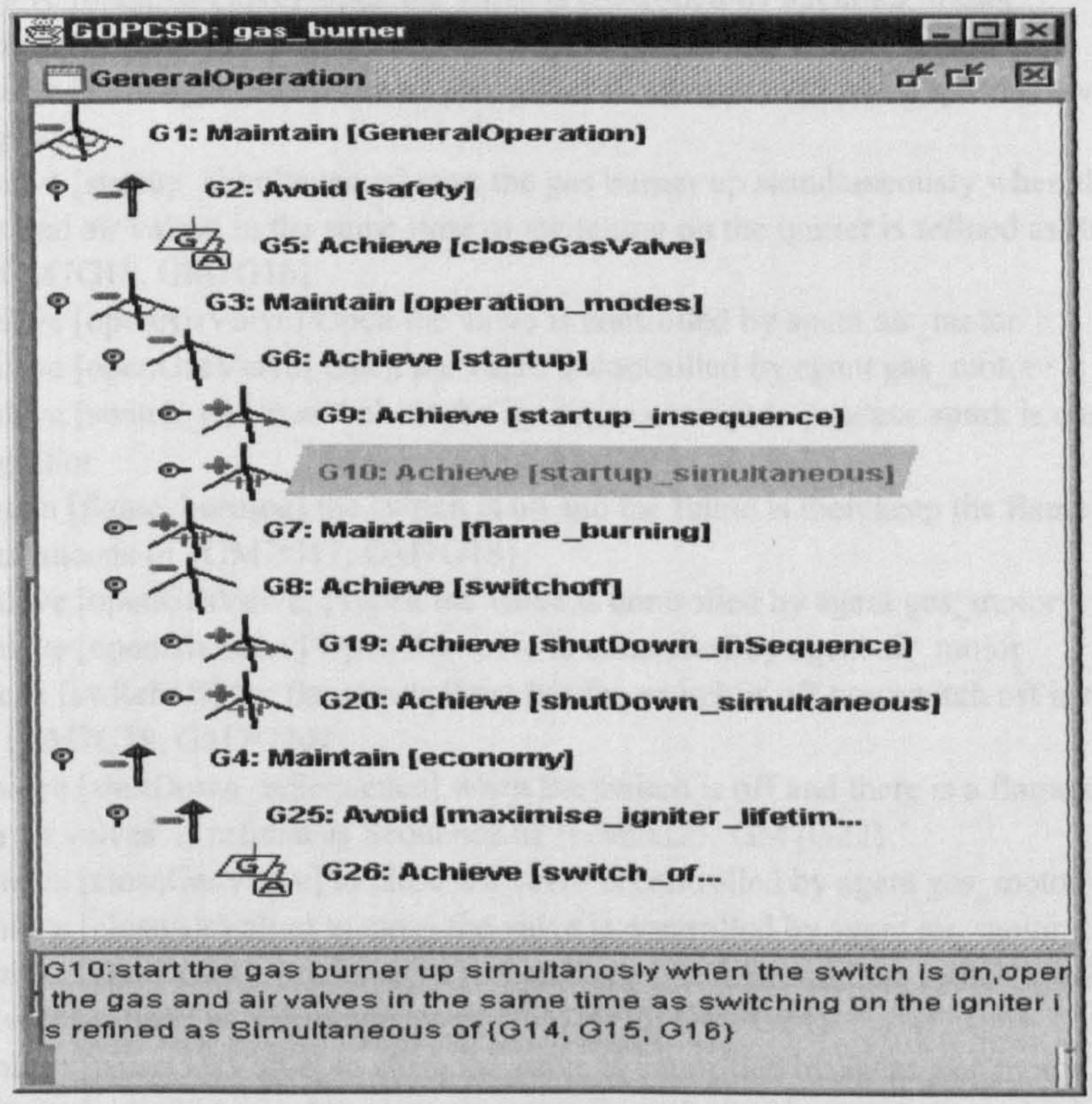


Figure 7.12, combining the safety, economy and operation goals in goal-model 7

After the user produces this goal-model he can delete goal-models 8 and 9, or keep them. In this case study, we preferred to remove them; hence, any reference to goal model 9 or 8 after this point will refer to two of the generated goal-models after splitting goal-model 7.

Before splitting the goal model, the user should attempt to remove any apparent problem or inefficiency. For this purpose, the tool enables a basic goal-model structure check to ensure that the

goal model does not violate the basic rules. In addition, the user can reason about the how and why of the goals and explore the variables' and agents' distribution over the goal-model (for more details, refer to chapter 6 section 6.2.8 and appendix A, section A.4, the tool user-guide.) In figure 7.13, an example of reasoning how for goal GM7G14 is displayed. It lists the ancestor goals' informal descriptions one by one, starting with the direct parent.

Table 7.6, the goals of goal-model 7

Goal-model 7 GeneralOperation :describe generally the high level functions of the burner
GM7G1: Maintain [GeneralOperation] general functions is refined as Conjunction of {GM7G2, GM7G3, GM7G4} GM7G2: Avoid [safety] avoid the gas valve open when the air valve is closed is refined as Inheritance of {GM7G5} GM7G5: Achieve [closeGasValve] if the gas valve is open and air valve is closed, close gas valve is controlled by agent gas_motor GM7G3: Maintain [operation_modes] is refined as Disjunction of {GM7G6, GM7G7, GM7G8} GM7G6: Achieve [startup] the system was off when the command of switching on was given start up is refined as Alternative of {GM7G9, GM7G10} GM7G9: Achieve [startup_insequence] start up in sequence open the air valve, then the gas valve the switch on the igniter is refined as Sequence of {GM7G11, GM7G12, GM7G13} GM7G11: Achieve [openAirValve] Open the valve is controlled by agent air_motor GM7G12: Achieve [openGasValve] Open the valve is controlled by agent gas_motor GM7G13: Achieve [switch_on] to switch on the igniter to attempt to produce spark is controlled by agent igniter_ignitior GM7G10: Achieve [startup_simultaneous] start the gas burner up simultaneously when the switch is on open the gas and air valves in the same time as switching on the igniter is refined as Simultaneous of {GM7G14, GM7G15, GM7G16} GM7G14: Achieve [openAirValve] Open the valve is controlled by agent air_motor GM7G15: Achieve [openGasValve] Open the valve is controlled by agent gas_motor GM7G16: Achieve [switch_on] to switch on the igniter to attempt to produce spark is controlled by agent igniter_ignitior GM7G7: Maintain [flame_burning] the switch is on and the flame is therekeep the flame burning is refined as Simultaneous of {GM7G17, GM7G18} GM7G17: Achieve [openGasValve,] Open the valve is controlled by agent gas_motor GM7G18: Achieve [openAirValve] Open the valve is controlled by agent air_motor GM7G8: Achieve [switchoff] the flame was there but the switch is off nowswitch off is refined as Alternative of {GM7G19, GM7G20} GM7G19: Achieve [shutDown_inSequence] when the switch is off and there is a flame detectedshut the gas and teh air valves is refined as Sequence of {GM7G21, GM7G22} GM7G21: Achieve [closeGasValve] to close the valve is controlled by agent gas_motor GM7G22: Achieve [closeAirValve] to close the valve is controlled by agent air_motor GM7G20: Achieve [shutDown_simultaneous] when there is a flame and the switch is off shut down gas and air valves is refined as Simultaneous of {GM7G23, GM7G24} GM7G23: Achieve [closeGasValve] to close the valve is controlled by agent gas_motor GM7G24: Achieve [closeAirValve] to close the valve is controlled by agent air_motor GM7G4: Maintain [economy] if the flame exists and switch is ON economical goal is refined as Inheritance of {GM7G25} GM7G25: Avoid [maximise_igniter_lifetime] if there is a flame present and the switch is onswitch off the spark ignitior is refined as Inheritance of {GM7G26} GM7G26: Achieve [switch_off] switch off the sparkswitch off the spark ignitor is controlled by agent igniter_ignitior

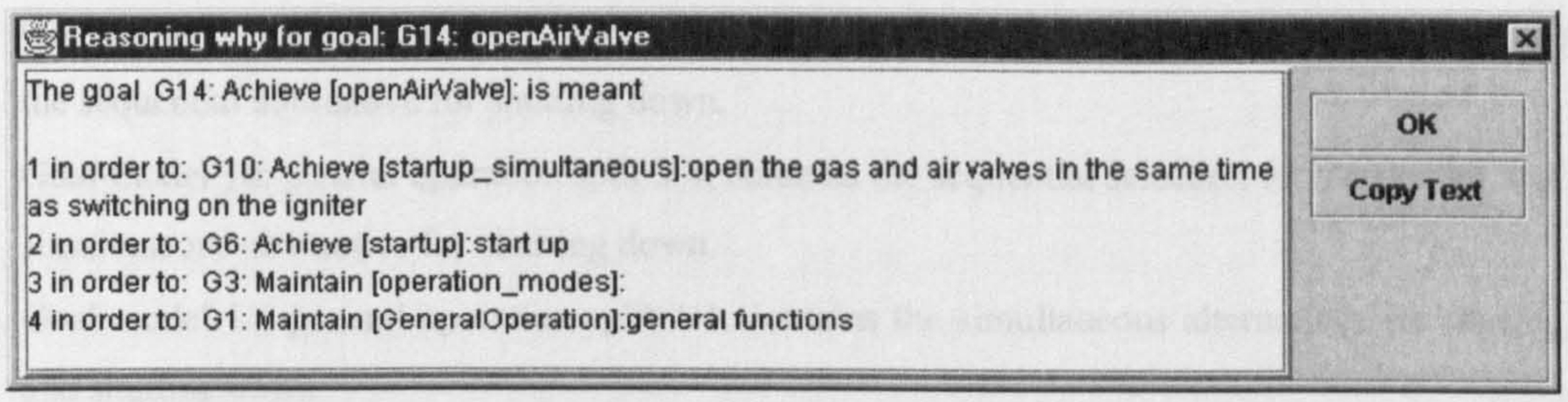


Figure 7.13, reasoning why about one of the goals

7.2.6 Splitting the goal-model

At this stage, goal-model 7 addresses the entire application. However, it still has alternative refinement sites at start up and shut down goals, where each of them has two alternative sub-goals: either sequential or simultaneous. This means that goal-model 7 is a compound solution. It will not be possible to perform consistency or completeness checks directly before splitting it into a list of simple solutions.

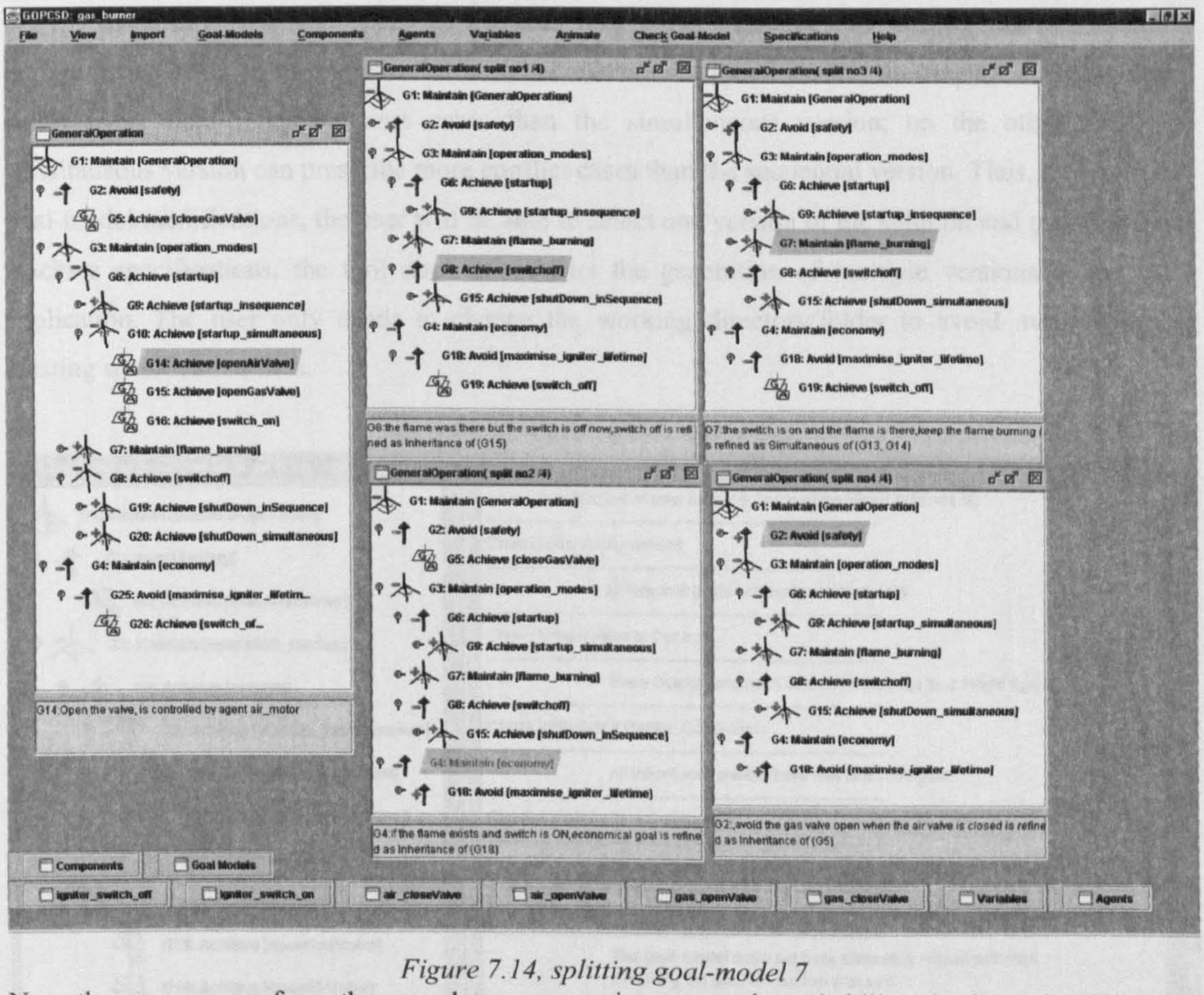


Figure 7.14, splitting goal-model 7

Now the user can perform the completeness, consistency, and reachability checks and validate the requirements expressed in the generated goal-models. There are four generated goal-models. In goal-model 7, there are two independent alternative goals; each of them has two sub-goals (one for sequential and one for simultaneous).

- Goal-model 8, general operation split 1/4, contains the sequential alternatives for starting up and shutting down.

- Goal-model 9, general operation split 2/4, contains the simultaneous alternative for starting up and the sequential alternative for shutting down.
- Goal-model 10, general operation split 3/4, contains the sequential selection for starting up and the simultaneous alternative for shutting down.
- Goal-model 11, general operation split 4/4, contains the simultaneous alternatives for starting up and shutting down.

Thus, it is more convenient for the user to check the first and the last solution versions because both of them are homogenous (the shutting down and starting up alternative are picked to be the same). In other applications, the user himself/herself can choose which versions are more suitable to proceed with. We assume the user deletes the second and third alternative; therefore, goal-model 8 now corresponds to the sequential version and goal-model 9 to the simultaneous version.

7.3 Checking and validating the requirements

Goal-models 8 and 9 can be checked now either separately or in turn; the modifications required may differ from one case to the other. As we discussed earlier, the sequential version may suffer from more incompleteness cases than the simultaneous version; on the other hand, the simultaneous version can prescribe more conflict cases than the sequential version. Thus, after possible goal-model modifications, the user will be able to select one version of the solution and generate the B machine specifications; the tool does not restrict the generation of multiple versions of the same application. The user only needs to change the working directory/folder to avoid overwriting the existing specification files.

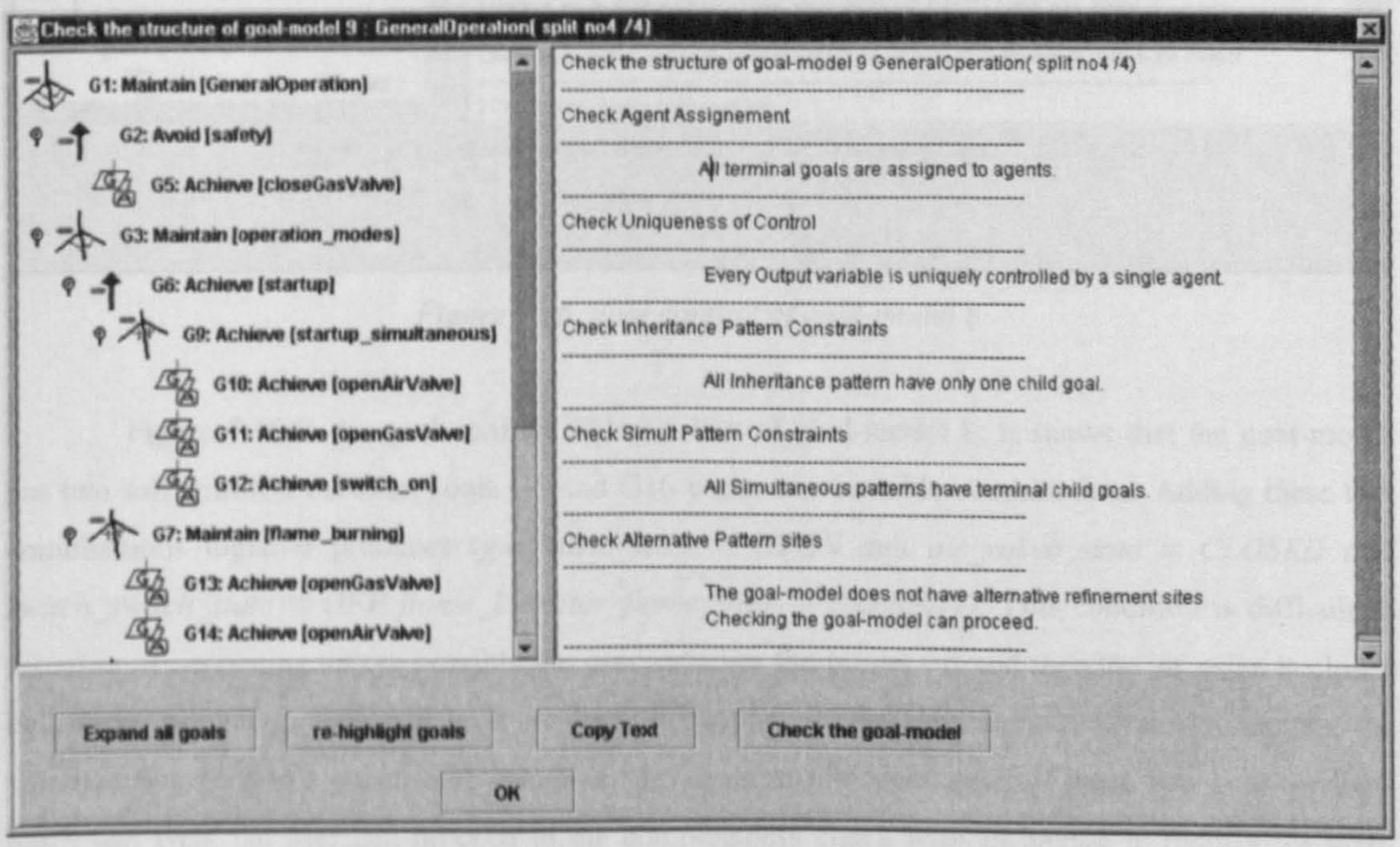


Figure 7.15, checking the structure of goal model 9

7.3.1 Goal-model structure check

Before checking the formal requirements details, the GOPCSD tool enables the user to ensure the goal-model structure is correct; since goal-model 7 has just been checked before the splitting process, there should not be any violation in the generated goal models 8 or 9, since the splitting process preserves the goal-model structure. Figure 7.15 shows the result of checking the correctness of the goal-model 9 structure.

7.3.2 Goal-conflict analysis

After the user ensures the goals of the goal-model are structured correctly and do not violate the basic constraints, the validation tests can proceed to the formal description to detect possible existing inconsistency within the goal-model in the form of conflict between goals that control a single output variable simultaneously.

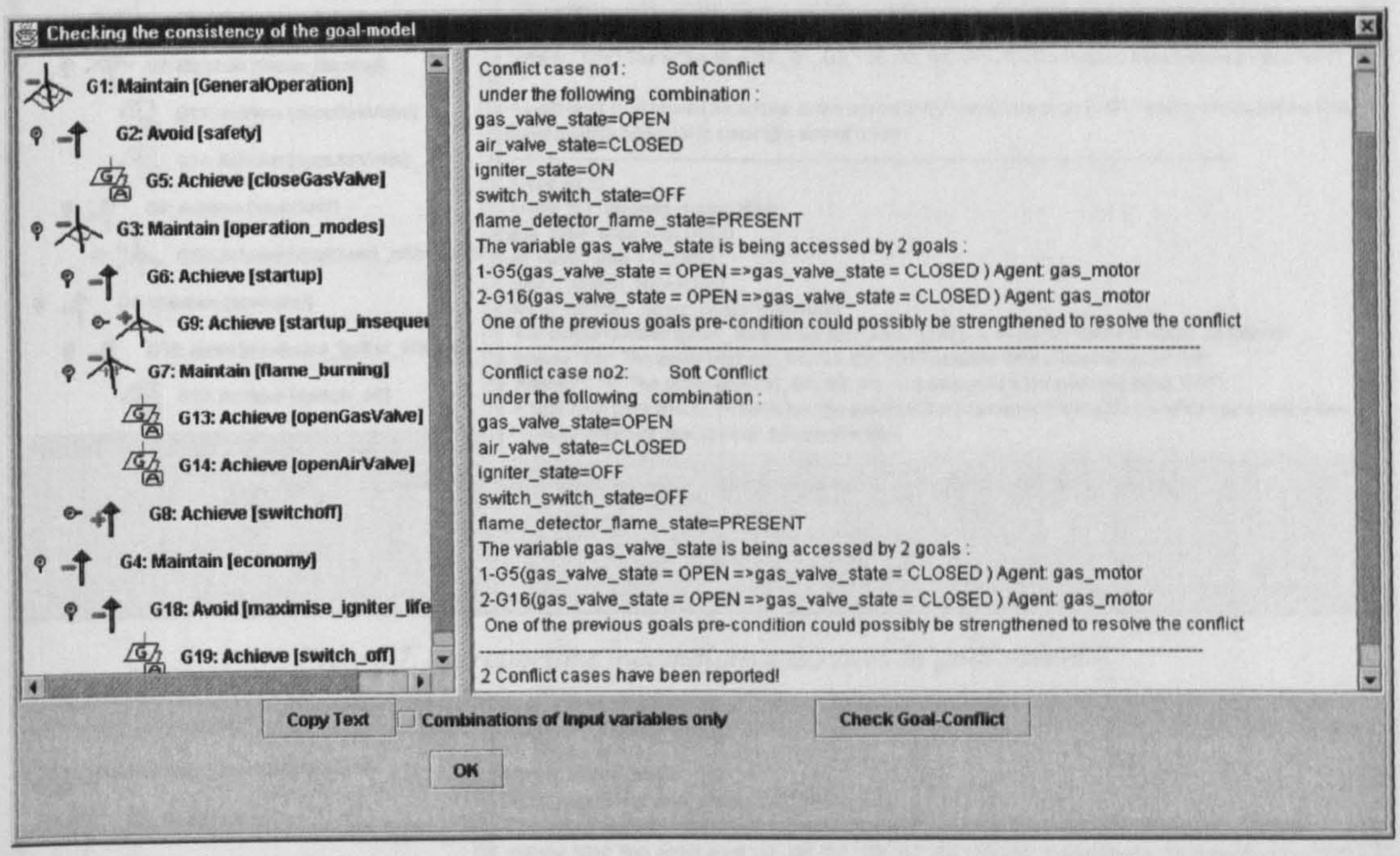


Figure 7.16, goal conflict of goal-model 8

Figure 7.16 is the goal-conflict dialogue box of goal-model 8; it shows that the goal-model has two soft conflicts between goals G5 and G16 under two variable combinations. Adding these two combinations together produces (*gas_valve_state = OPEN and air_vakve_state = CLOSED and switch_switch_state = OFF flame_Detector_flame_state = PRESENT)*. This condition is difficult to visualise as happening unless possibly the user switches the burner off and then the air valve is closed before the gas valve. This will be more likely to happen in the simultaneous version. Checking the simultaneous version's consistency produces the same result. Since each of these two goal-conflicts has a soft type, the user can proceed to the completeness check without having to modify the goal-model.

7.3.3 Checking the Completeness and Reachability

The next step is to check the completeness and reachability of the goal-model. We performed the goal-reachability check on goal models 8 and 9 and both have no unreachable goals. Then we performed the completeness check to address the issue of covering the possible situations as well as the determinism in covering them. As shown in figures 7.17 and 17.18, the GOPCSD discovered 21 incompleteness cases reported in the sequential version and 18 cases in the simultaneous version, respectively (see appendix B for more details).

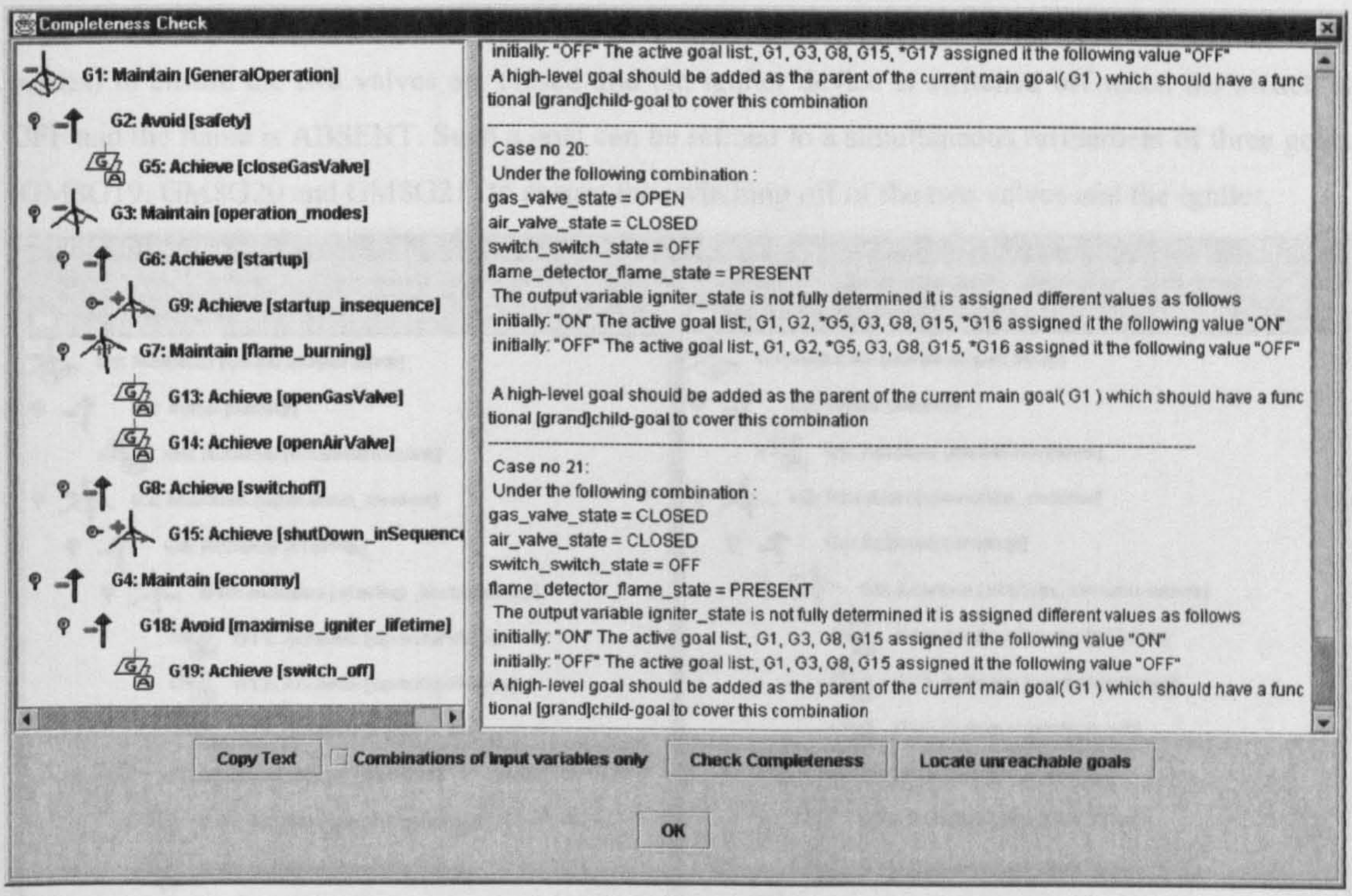


Figure 7.17, reporting incompleteness cases of goal model 8

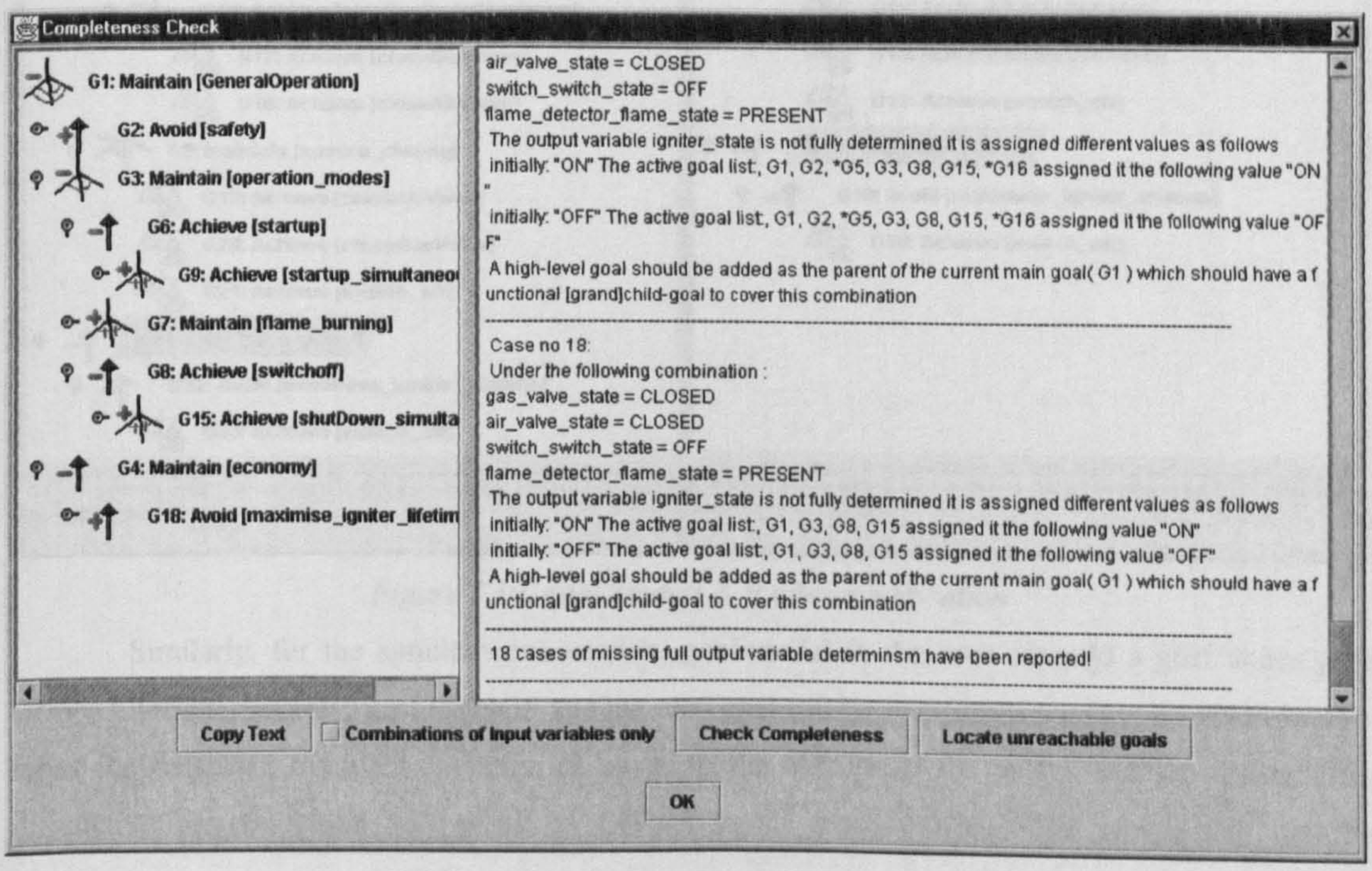


Figure 7.18, checking the completeness of goal model 9

7.3.4 Modifying the goal-models and repeating the checks

Looking at the results of the completeness check, one can identify one pattern of incompleteness when there is a lack of control; this appears when the output variables' values do not change from their initial value before executing the goal-model for one cycle, as shown in case 20 listed in figure 7.17. Looking through these cases can guide the user to create a new goal to ensure the switching off of the burner system. The displayed active goal lists within each incompleteness case should provide some guidance on where to place this goal. For goal-model 8, the sequential version, the user can add a new compound goal (ensure closing, GM8G9) under goal GM8G3 (operation modes) to ensure the two valves are closed and the igniter device is switched off when the switch is OFF and the flame is ABSENT. Such a goal can be refined to a simultaneous refinement of three goals (GM8G19, GM8G20 and GM8G21) to ensure the switching off of the two valves and the igniter.

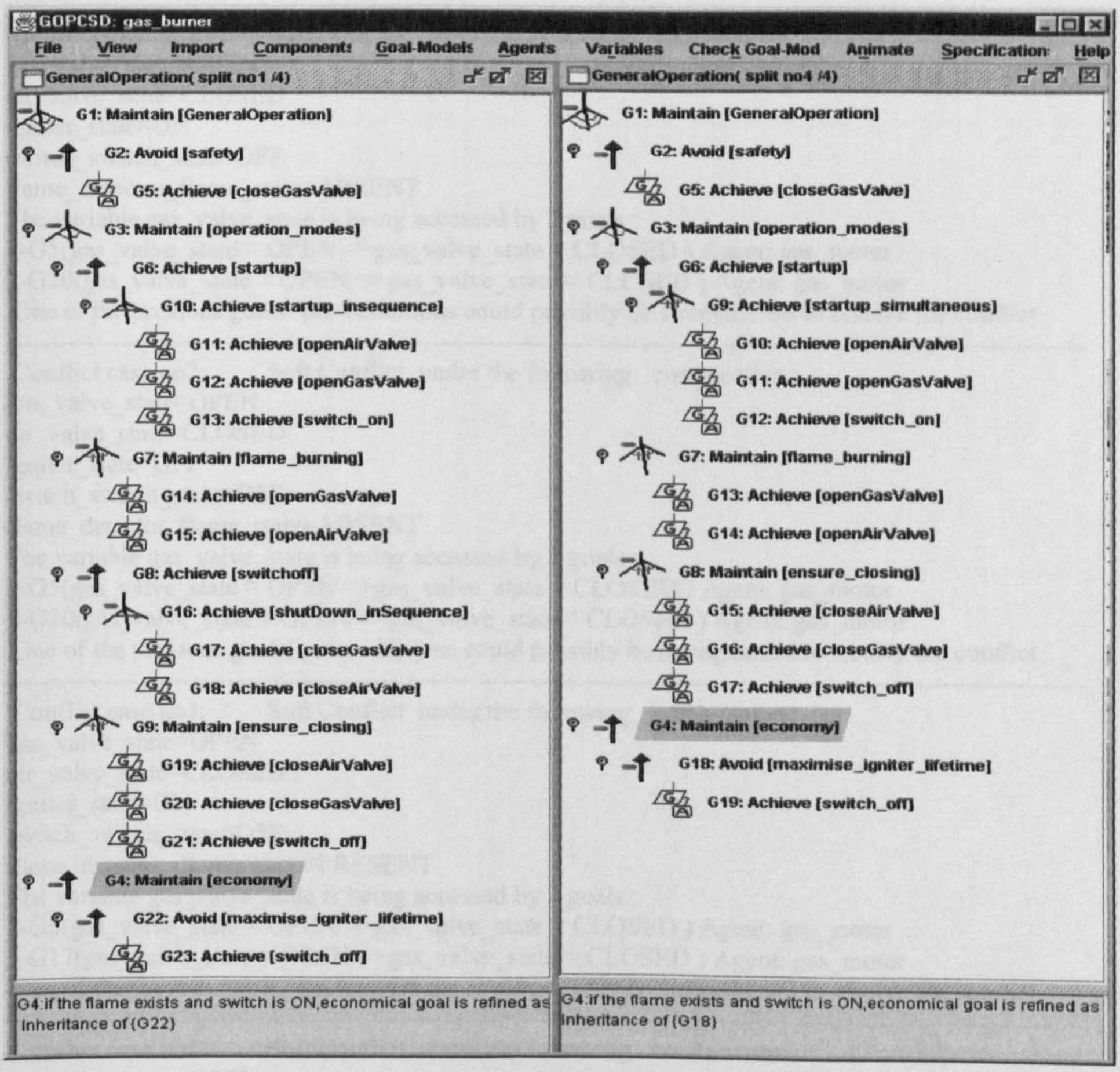


Figure 7.19, goal-models 8, 9 after modification

Similarly, for the simultaneous version, goal-model 9, the user can add a goal under goal GM9G3. Or alternatively, goal GM9G8 and the new goal can be compacted together to cover the two cases for switching off after operation or ensuring the closure of the valves and the igniter. The disjunctive pre-conditions will result in (*switch_switch_state =OFF*). The compacting can be

performed by adding a new sub-goal to the shut down simultaneously goal GM9G15, and, then, optionally moving the goal up one level instead of the parent goal GM9G8.

If the user needs to modify the goal-model in different ways, he/she can copy a new version of the same goal-model and modify it in a new workspace or again he can use the alternative pattern refinements and then split the compound goal-model and test the different modified versions. Figure 7.19 shows goal-models 8 and 9 after applying these modifications.

After modifying the goal-models, the user should check their structural correctness, check their consistency, and finally he/she can repeat the completeness check again. Both of the goal-model versions passed the valid structure check. However, each of them suffered from four goal-conflict cases listed as follows.

```
Goal-Conflict check: goal-model goal models 8,9 (different goal names but same situations)
----------
Conflict case no1:      Soft Conflict  under the following   combination :
gas_valve_state=OPEN
air_valve_state=CLOSED
igniter_state=ON
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
The variable gas_valve_state is being accessed by 2 goals :
1-G5(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
2-G20(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
Conflict case no2:      Soft Conflict  under the following   combination :
gas_valve_state=OPEN
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
The variable gas_valve_state is being accessed by 2 goals :
1-G5(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
2-G20(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
Conflict case no3:      Soft Conflict  under the following   combination :
gas_valve_state=OPEN
air_valve_state=CLOSED
igniter_state=ON
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
The variable gas_valve_state is being accessed by 2 goals :
1-G5(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
2-G17(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
Conflict case no4:      Soft Conflict  under the following   combination :
gas_valve_state=OPEN
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
The variable gas_valve_state is being accessed by 2 goals :
1-G5(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
2-G17(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
```

```
4 Conflict cases have been reported!
```

The above listed cases reveal the following facts:

- Even though the goal hierarchies of goal models 8 and 9 are not exactly the same; the reported conflict situations are the same.
- The consistency of the goal-models did not suffer from adding the "ensuring closed valves and igniter" goal.
- Each of these goal-conflict cases has goal G5 (close the gas valve if the air is open) as one of its conflicting goals.
- Oring the conditions of all these cases together results in[1] *(gas_valve_state=OPEN and air_valve_state=CLOSED and switch_switch_state=OFF)*. If the user wants to get rid of these conflict cases, he/she should possibly restrict goal GM9G5 or GM8G5 to operate only when the switch is ON (i.e. the gas valve is witched off under the operating goals if the switch is OFF or under an unsafe situation if the switch is OFF).

We will assume that the user did not modify the goal-models since they contain only soft conflicts. The check can proceed now by performing the completeness test on goal-models 8 and 9. For the simultaneous version goal-model 9, only four incompleteness cases are reported as follows.

```
Completeness check: goal-model 9
---------------------------------------------------------------------------------------
Case no 1: Under the following combination :
air_valve_state = CLOSED
igniter_state = ON
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G6, G9, *G10 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G6, G9, *G10, *G11 assigned it the following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case
---------------------------------------------------------------------------------------
Case no 2: Under the following combination :
air_valve_state = CLOSED
igniter_state = OFF
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G6, G9, *G10, *G12 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G6, G9, *G10, *G11, *G12 assigned it the following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case
---------------------------------------------------------------------------------------
Case no 3: Under the following combination :
air_valve_state = CLOSED
igniter_state = ON
switch_switch_state = ON
flame_detector_flame_state = PRESENT
```

[1] We used $(X \wedge \neg Y \vee X \wedge Y \equiv X)$ to combine the four cases.

```
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G14, G4, G18, *G19 assigned it the
following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G13, *G14, G4, G18, *G19 assigned it the
following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case
-------------------------------------------------------------------------------------------------------
Case no 4: Under the following combination :
air_valve_state = CLOSED
igniter_state = OFF
switch_switch_state = ON
flame_detector_flame_state = PRESENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G14, G4, G18 assigned it the following
value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G13, *G14, G4, G18 assigned it the
following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case
-------------------------------------------------------------------------------------------------------
4 cases of missing full output variable determinism have been reported!
-------------------------------------------------------------------------------------------------------
```

Unlike the earlier identified pattern, lack of control, these incompleteness cases prescribe a kind of inconsistency that allows the initial value of the output variable to determine its final value. In this sense, these cases can be resolved or ignored according to the user's perspective. Oring the conditions of these four cases results in *(switch_switch_state=ON and air_valve_state = CLOSED),* the case when the air valve is closed and the switch is on. If the gas valve is closed, then normally the system will open it from the operational aspect. On the other hand, if the gas valve is already open (unsafe situation) the system will close the valve to maintain safe conditions. This can result in an oscillatory behaviour; the gas valve will alternate between opening and closing states, if the other conditions remained constant. It is essential to remove such a situation, which arose because of goal GM9G5. On the one hand it will close the valve if it is open; on the other, goals GM9G11 (Open the gas valve for starting up) or GM9G13 (Keep opening the gas valve to maintain the flame burning) will open the valve if it is closed. Therefore, apart from the gas valve state, if the pre-conditions of these goals are exclusive, the behaviour will be deterministic.

This can be achieved by enabling the safety goal GM9G5 to close the valve only when the switch is off. This is probably unacceptable because the unsafe situation can happen during starting up. The other choice that is logical is that the gas valve will not be open unless the air valve is already open, i.e. to restrict goals GM9G11 and GM9G13 by adding *(air_valve-state=OPEN)* in their pre-conditions.

Having performed these modifications, the user himself/herself forced the safety constraints on the operational goals. An expert user could formulate goals GM9G11 and GM9G13 considering the air valve, but, this shows an example of how the second phase of the GOPCSD tool can help the user to correct the requirements and remove logical errors as early as possible.

Applying the completeness and reachability check to goal-model 8 (the sequential version) produced the following 11 incompleteness cases:

```
Completeness check: goal model 8
-------------------------------------------------------------------------------------------------------
Case no 1: Under the following combination :
```

air_valve_state = CLOSED
igniter_state = ON
switch_switch_state = ON
flame_detector_flame_state = PRESENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G15, G4, G22, *G23 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G14, *G15, G4, G22, *G23 assigned it the following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 2: Under the following combination :
air_valve_state = CLOSED
igniter_state = OFF
switch_switch_state = ON
flame_detector_flame_state = PRESENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G15, G4, G22 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G14, *G15, G4, G22 assigned it the following value "OPEN"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 3: Under the following combination :
gas_valve_state = OPEN
igniter_state = ON
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3, G8, G16, *G17 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3, G8, G16, *G17 assigned it the following value "CLOSED"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 4: Under the following combination :
gas_valve_state = OPEN
igniter_state = OFF
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3, G8, G16, *G17 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3, G8, G16, *G17 assigned it the following value "CLOSED"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 5: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = OPEN
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G6, G10, *G12 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G6, G10, *G12 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 6: Under the following combination :
gas_valve_state = OPEN
air_valve_state = CLOSED

switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G2, *G5, G3, G6, G10, *G11 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G2, *G5, G3, G6, G10, *G11 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 7: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = CLOSED
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G6, G10, *G11 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G6, G10, *G11 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 8: Under the following combination :
gas_valve_state = OPEN
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G16, *G17 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G8, G16, *G17 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 9: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G16, *G18 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G8, G16, *G18 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 10:
Under the following combination :
gas_valve_state = OPEN
air_valve_state = CLOSED
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G2, *G5, G3, G8, G16, *G17 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G2, *G5, G3, G8, G16, *G17 assigned it the following value "OFF"
Possibly, functional goals could be added under one of the listed goals in each case

---

Case no 11: Under the following combination:
gas_valve_state = CLOSED
air_valve_state = CLOSED
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G16 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G8, G16 assigned it the following value "OFF"

Possibly, functional goals could be added under one of the listed goals in each case

11 cases of missing full output variable determinism have been reported!

Two of these incompleteness cases (cases 1 and 2) have the "counter action" pattern between the safety goal GM8G5 and goal GM8G14 that keeps the gas valve open to maintain the flame burning; it can be removed by the same treatment as in the simultaneous version. One can notice here that goal GM8G9 does not prescribe counter action against the safety goal GM8G5 because it is already pre-conditioned by the post-condition of its predecessor goal GM8G10, which has the post condition of *(air_valve_state=OPEN)*.

The other nine reported cases have the "lack of control" pattern (that can appear from the active list, which has no goals marked with * to show a terminal goal that changes the output variables) and can be grouped as follows:

- Cases 3 and 4: lack of control of the air valve when the gas valve is OPEN, the switch is OFF and the flame is PRESENT. This may be ignored as the active list shows that if the air valve is CLOSED, either one of the goals GM8G5 and GM8G17 will close the gas valve; otherwise GM8G17 will close the gas valve to maintain the safety condition.
- Cases 5, 6 and 7: lack of control of the igniter if the switch is ON and flame is ABSENT and at least one of the two valves is CLOSED. This means if the gas or air valve is closed, the igniter is not controlled. It may be treated or not according to the user perspective. If the user requires the system to only switch on the igniter when the two valves are open, then he/she had better state that the igniter should be switched off otherwise. A little consideration can show weakening the condition of the economy goal that switches the igniter off can remove these cases.
- Cases 8, 9, 10 and 11: lack of control of the igniter if the flame is PRESENT and the switch is OFF. Again these cases can be removed by omitting the *(switch_switch_state =ON)* from the economy goal pre-condition.

Thus, to cover cases 5, 6, 7, 8, 9, 10 and 11, one can change the pre-condition of the economy goal GM8G23 or one of its parent goals to (*flame_detector_flame_state = PRESENT or (flame_detector_flame_state = ABSENT and (gas_valve_state= CLOSED or air_valve_state = CLOSED)))* or using the absorption property in Boolean logic it can be simplified to *(flame_detector_flame_state = PRESENT or gas_valve_state = CLOSED or air_valve_state = CLOSED)*, which could be easily specified by the user if there is no flame or one of the valves are not open.

After modifying goal-models 8 and 9 to remove the incompleteness cases, the goal-model structure, goal-conflict, goal-reachability and completeness checks should be performed once again, before animating the goal-models.

Goal-model 9 now has four soft goal-conflict cases, no unreachable-goals and no incompleteness cases. Goal-model 8 has four soft goal-conflicts, no unreachable goals, and 2 incompleteness cases (cases 3 and 4). The user can animate them now and modify them to remove any logical errors detected through the animation.

One can notice that the GOPCSD tool helped the user to correct the initial imperfect design. In this respect, it did not force him to have a complete and consistent goal model by phase one, but rather guided him to reach this state before proceeding to phase three.

7.3.5 Obstacle analysis

The GOPCSD tool enables the user to document the obstacles, which may happen during run time to obstruct the achievement of some of the goals. Considering these obstacles usually extends the domain of the application. This extension can be regarded as adding segments of the environment's details to the studied application. For example, some obstacles can arise because of temporal disconnection between the power and control circuits. The tool enables the user to mark the obstructed goals and then report the possible treatment that can be applied separately for each obstruction case.

7.3.6 Animating the Requirements

Validating the goal-model is an important step, especially after the user modified the goal-model after performing the conflict and the completeness checks. In addition, the animation can provide an essential means to prefer one final version over another. The user can animate the two goal-models in turn to judge the relevant attributes used to choose amongst solutions; in addition, he/she can test application reliability against output changes and unexpected events. As shown in figure 7.21, goal-model 9 is displayed to the left in the dialogue box; this enables the user to observe the correlation between the goals and the animation result. During the animation cycles, the active branches are highlighted to help the user to locate the active goals.

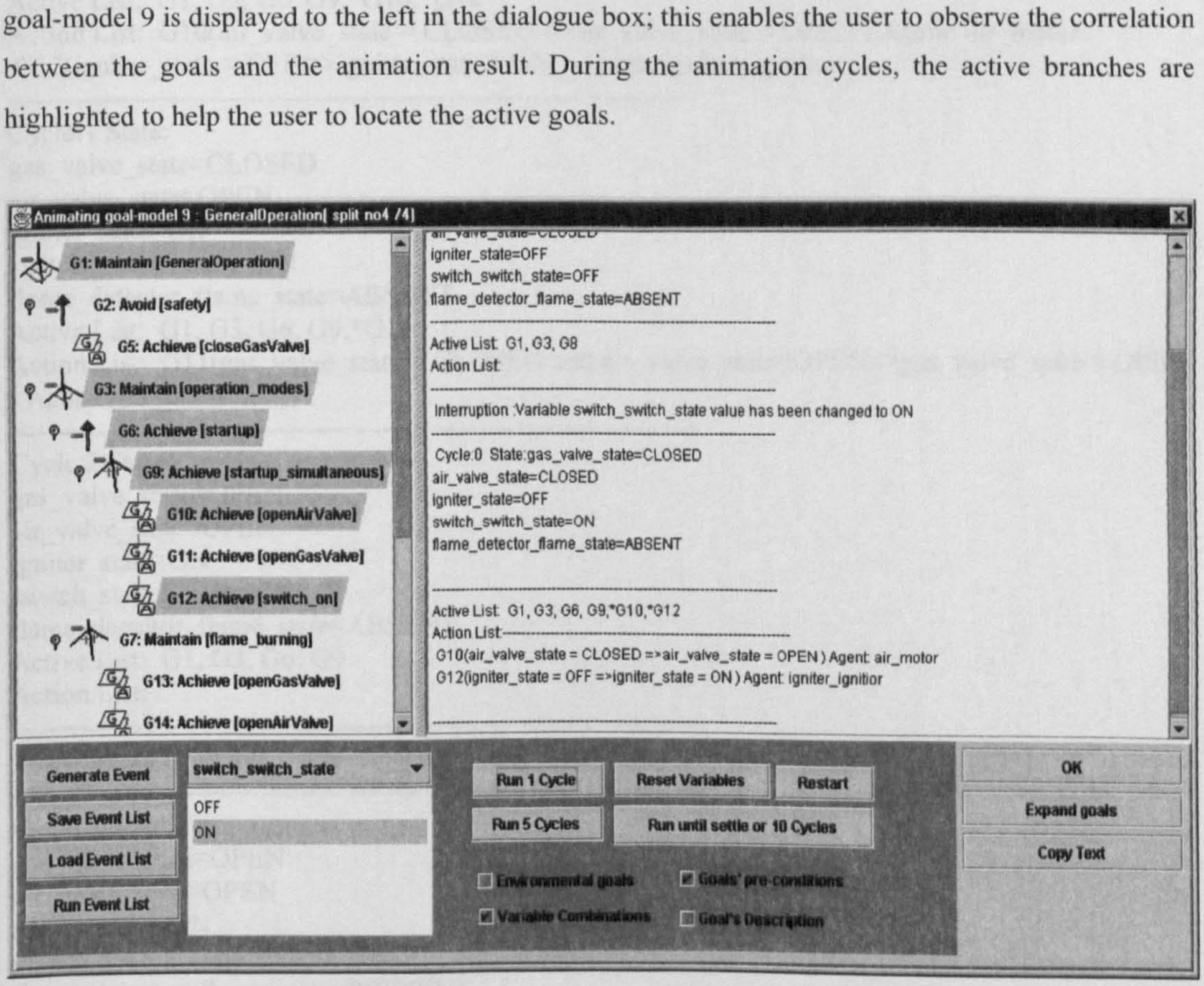


Figure 7.20, the animation of goal-model 9, the simultaneous version.

Animating goal-model 9 starts by initialising the variables and then simulating the start up event by changing the *switch_switch_state* to be ON. Then, the tool executes the goal-model cycle by cycle; in each cycle, the activated goals are listed as well as the details of the terminal goals, which change the output variables. Moreover, the active list of the non-terminal goals with satisfied pre-conditions and action list of the terminal goals with satisfied pre-conditions are highlighted in the goal-model to the left. The following is a listing of normal starting up of the burner, followed by examining the system response in the case when the flame went off, then shutting down.

```
Animating goal-model 9 GeneralOperation( split no4 /4)
 Resetting all Variables to the initial values:
---------------------------------------------------------------------
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
---------------------------------------------------------------------
 Interruption :Variable switch_switch_state value has been changed to ON
---------------------------------------------------------------------
Cycle:0  State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9,*G10,*G12
Action List:  G10(air_valve_state = CLOSED =>air_valve_state = OPEN ) Agent: air_motor
 G12(igniter_state = OFF =>igniter_state = ON ) Agent: igniter_ignitior
---------------------------------------------------------------------
Cycle:1 State:
gas_valve_state=CLOSED
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9,*G11
Action List:  G11(gas_valve_state = CLOSED and air_valve_state=OPEN=>gas_valve_state = OPEN
) Agent: gas_motor
---------------------------------------------------------------------
Cycle:2 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9
Action List:
---------------------------------------------------------------------
 Interruption :Variable flame_detector_flame_state value has been changed to PRESENT
---------------------------------------------------------------------
Cycle:3 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G7, G4, G18,*G19
Action List:  G19(igniter_state = ON =>igniter_state = OFF ) Agent: igniter_ignitior
```

Cycle:4 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List: G1, G3, G7, G4, G18
Action List:

Interruption :Variable flame_detector_flame_state value has been changed to ABSENT

Cycle:5 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List: G1, G3, G6, G9,*G12
Action List: G12(igniter_state = OFF =>igniter_state = ON) Agent: igniter_ignitior
---
Cycle:6 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List: G1, G3, G6, G9
Action List:

Interruption :Variable flame_detector_flame_state value has been changed to PRESENT

Cycle:7 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List: G1, G3, G7, G4, G18,*G19
Action List: G19(igniter_state = ON =>igniter_state = OFF) Agent: igniter_ignitior
---
Cycle:8 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List: G1, G3, G7, G4, G18
Action List:

Interruption :Variable switch_switch_state value has been changed to OFF

Cycle:9 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
Active List: G1, G3, G8,*G15,*G16
Action List: G15(air_valve_state = OPEN =>air_valve_state = CLOSED) Agent: air_motor

```
 G16(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
---------------------------------------------------------------------
Cycle:10 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G8
Action List:
---------------------------------------------------------------------
 Interruption :Variable flame_detector_flame_state value has been changed to ABSENT
---------------------------------------------------------------------
Cycle:11 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G8
Action List:
```

It is important to allow the user to observe the possible behavioural change from the initial plans; for example, the starting up in this version was meant to be achieved simultaneously; however to maintain safe conditions, the gas valve will be open only after the air valve. This appears in cycles 1 and 2, where the air valve is open and the igniter is switched on, then in the next cycle the gas valve is open.

Another useful usage of the animation is to inspect the unsafe situation where the system is usually required to recover to a safe state; the following animation shows an example of the unsafe state when the gas valve is open and the air valve is closed.

```
Animating goal-model 9 GeneralOperation( split no4 /4)
 Reseting all Variables to the initial values:
---------------------------------------------------------------------
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
---------------------------------------------------------------------
 Interruption :Variable gas_valve_state value has been changed to OPEN
---------------------------------------------------------------------
Cycle:0 State:
gas_valve_state=OPEN
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
Active List:  G1, G2,*G5, G3, G8,*G16
Action List:  G5(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
 G16(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
---------------------------------------------------------------------
Cycle:1 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
```

```
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G8
Action List:
-----------------------------------------------------------------------
```

The system recovered to a state where the gas valve was closed.

Figure 7.21 represents a state transition diagram STD of the gas burner system. We identified stable and transient states; when the gas burner is in one of the stable states it does not change its state unless it receives an external stimulus or a sudden change happen to one of the output variables. The transient states are reached after stimulating the system and each of them has at least an active terminal goal that will change the gas burner entire state. The continuous arrows represent external stimuli, like Switch = ON or Flame_Detected = PRESENT; the dotted arrows are labelled by the terminal goal(s) which is activated from the transient state and, by achieving its post condition the system changes its state; finally, the bold arrows represent a sudden change in the output variables, which can be used to represent faults and response delay.

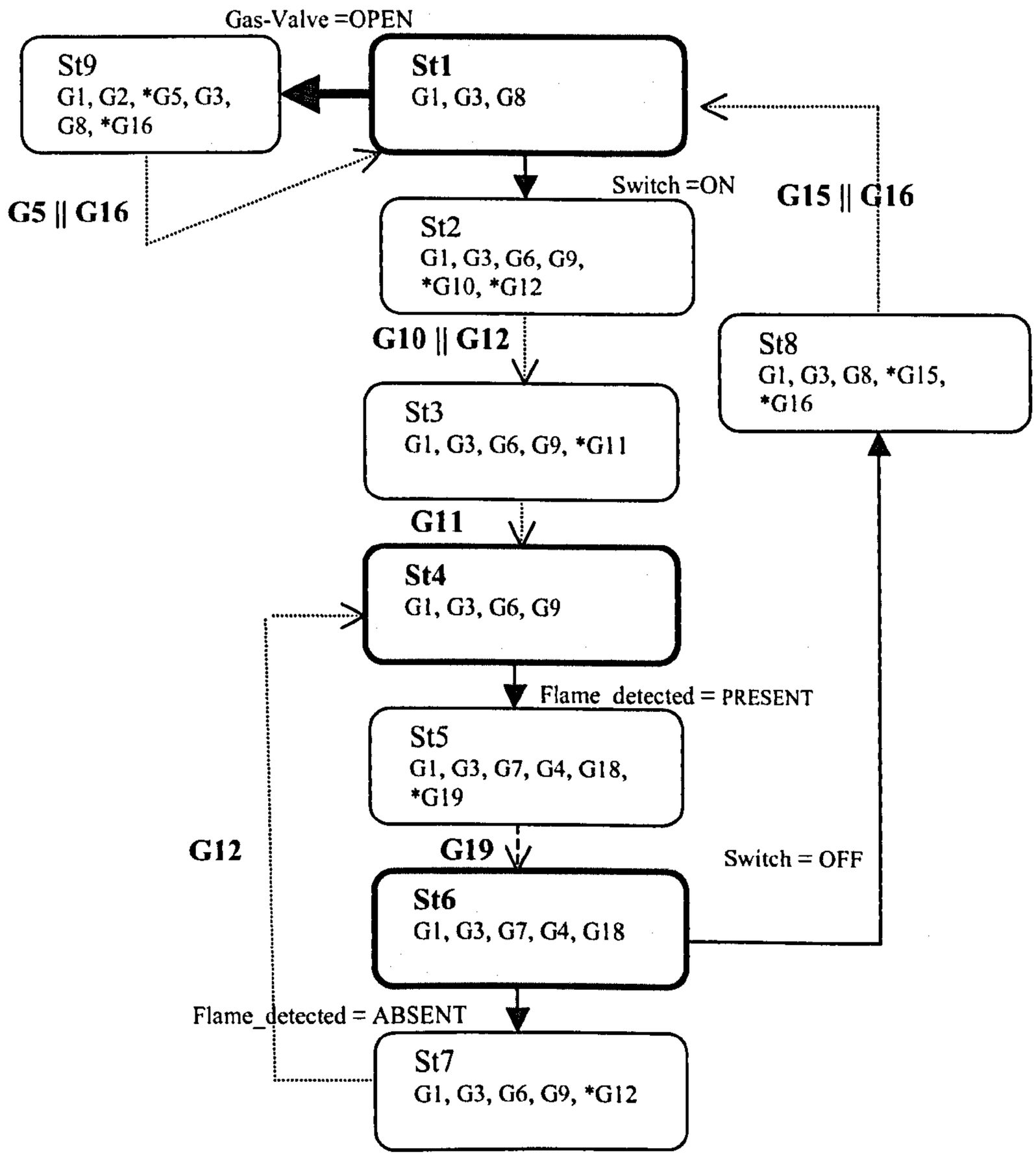


Figure 7.21, constructing a STD from the goal-model 9 animation data

This illustrates that the terminal goals are represented as the direct transition from one state to another, while the non-terminal goals are represented by indirect transitions, which are a compound of parallel and consecutive direct transitions.

State *st1* is the initial state; the system moves to state *st2* when the switch is changed to ON. In the transient state *st2,* the gas burner attempts to open the air valve (*G10) and produce a spark (*G12). When it succeeds, the state changes to another transient state, s*t3,* where it attempts to open the gas valve (after insuring the air valve is open). Having opened the gas valve, the system reaches a stable state, *st4*. State st4 represents the situation of the gas burner system when the two valves are open and the system attempts to produce the spark; having detected a flame, the state changes to the transient state *st5*, where the system attempts to increase the lifetime of the igniter by switching the igniter off. After the igniter is switched off, the system reaches a stable state, *st6*. If for any reason the flame goes off, the system will go to transient state *st7*, where it attempts to produce a spark; achieving this, the system will remain in state *st4*. When the system is switched off from state *st6*, it attempts to close the two valves from within state *st8;* then it reaches the initial idle state *st1*. As an example to test an unsafe condition if the gas valve is opened for some reason, the system goes to state *st9,* where it attempts to close the gas valve to ensure the safety conditions.

The two states *st2* and *st7*, where each of them has two terminal goals, may require further animation especially to acquire the user's satisfaction in the case when one of the goals is achieved before the others. For example, the animation utility can be used to inspect the case when goal G10 has been achieved before goal G12.

The animation utility enables the user to invasatgae the fine grain view of such transient states as state *st2* in figure 7.21; in order to explore the case when opening the air valve takes further time than producing the spark, the user can force the value of the air motor to be CLOSED, as shown by the following animation data.

```
Animating goal-model 9 GeneralOperation( split no4 /4)
 Resetting all Variables to the initial values:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
---------------------------------------------------------------------
Cycle:0 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G8
Action List:
---------------------------------------------------------------------
 Interruption :Variable switch_switch_state value has been changed to ON
---------------------------------------------------------------------
Cycle:1 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=ON
```

```
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9,*G10,*G12
Action List:  G10(air_valve_state = CLOSED =>air_valve_state = OPEN ) Agent: air_motor
 G12(igniter_state = OFF =>igniter_state = ON ) Agent: igniter_ignitior
---------------------------------------------------------------------
 Interruption :Variable air_valve_state value has been changed to CLOSED
---------------------------------------------------------------------
Cycle:2 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9,*G10
Action List:  G10(air_valve_state = CLOSED =>air_valve_state = OPEN ) Agent: air_motor
---------------------------------------------------------------------
Cycle:3 State:
gas_valve_state=CLOSED
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G9,*G11
Action List:  G11(gas_valve_state = CLOSED and air_valve_state=OPEN=>gas_valve_state = OPEN
) Agent: gas_motor
```

Similarly, the user can investigate the case when the air valve opens before the spark is produced. The STD shown in figure 7.22 illustrates this fine-grain investigation.

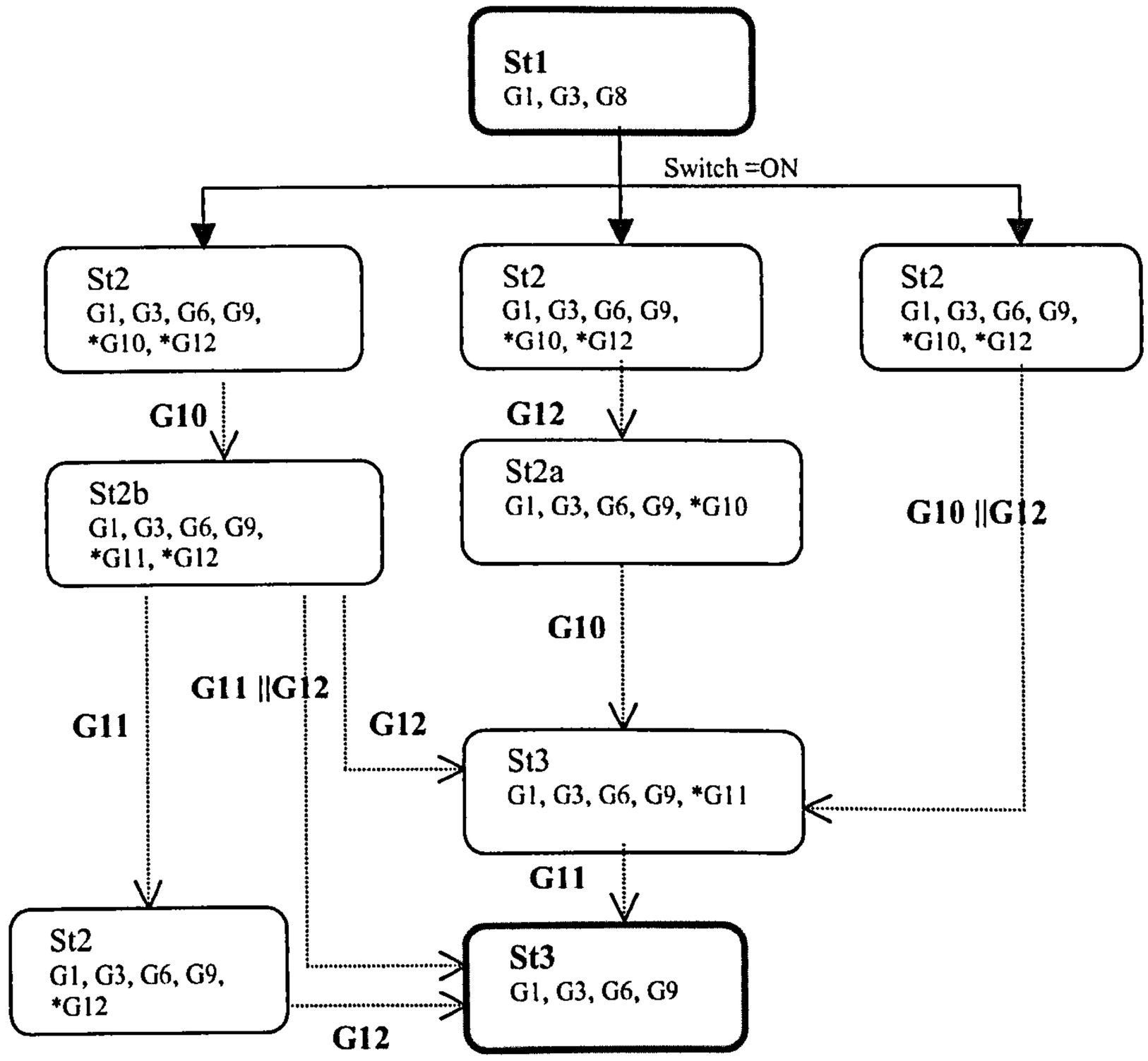


Figure 7.22, the fine grain behaviour of the transition from state st1 to st3

The transition from the stable state *st1* to *st3* can be achieved through one of the indicated paths, according to the relative response of the two valves and the igniter and obeying the safety constraint that the air value must be open (goal G10) before the gas valve can be open (goal G11).

This STD in the figure represents the normal cycle of the gas burner system; it is similar to the initial expected analysis in section 4.7; this clarifies some aspects of the GOPCSD method, which guides the systems engineer to achievw a specification of the correct requirement without being aware of statecharts or sophisticated logic or mathematics.

For the sequential version, goal-model 8, the same scenario was executed and the following results were collected:

```
Animating goal-model 8 GeneralOperation( split no1 /4)
 Resetting all Variables to the initial values:
---------------------------------------------------------------------
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=ABSENT
---------------------------------------------------------------------
 Interruption :Variable switch_switch_state value has been changed to ON
---------------------------------------------------------------------
Cycle:0 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10,*G11, G4, G22
Action List:  G11(air_valve_state = CLOSED =>air_valve_state = OPEN ) Agent: air_motor
---------------------------------------------------------------------
Cycle:1 State:
gas_valve_state=CLOSED
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10,*G12, G4, G22
Action List:  G12(gas_valve_state = CLOSED  and air_valve_state = OPEN =>gas_valve_state =
OPEN ) Agent: gas_motor
---------------------------------------------------------------------
Cycle:2 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10,*G13
Action List:  G13(igniter_state = OFF  and air_valve_state = OPEN  and gas_valve_state = OPEN
=>igniter_state = ON ) Agent: igniter_ignitior
---------------------------------------------------------------------
Cycle:3 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10
```

```
Action List:
----------
 Interruption :Variable flame_detector_flame_state value has been changed to PRESENT
----------
Cycle:4 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G7, G4, G22,*G23
Action List:  G23(igniter_state = ON =>igniter_state = OFF ) Agent: igniter_ignitior
----------
Cycle:5 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G7, G4, G22
Action List:
----------
 Interruption :Variable flame_detector_flame_state value has been changed to ABSENT
----------
Cycle:6 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10,*G13
Action List:  G13(igniter_state = OFF  and air_valve_state = OPEN  and gas_valve_state = OPEN
=>igniter_state = ON ) Agent: igniter_ignitior
----------
Cycle:7 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=ABSENT
Active List:  G1, G3, G6, G10
Action List:
----------
 Interruption :Variable flame_detector_flame_state value has been changed to PRESENT
----------
Cycle:8 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=ON
switch_switch_state=ON
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G7, G4, G22,*G23
Action List:  G23(igniter_state = ON =>igniter_state = OFF ) Agent: igniter_ignitior
----------
Cycle:9 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=ON
flame_detector_flame_state=PRESENT
```

```
Active List:  G1, G3, G7, G4, G22
Action List:
----------
 Interruption :Variable switch_switch_state value has been changed to OFF
----------
Cycle:10 State:
gas_valve_state=OPEN
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G8, G16,*G17, G4, G22
Action List:  G17(gas_valve_state = OPEN =>gas_valve_state = CLOSED ) Agent: gas_motor
----------
Cycle:11 State:
gas_valve_state=CLOSED
air_valve_state=OPEN
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G8, G16,*G18, G4, G22
Action List:  G18(air_valve_state = OPEN  and gas_valve_state = CLOSED =>air_valve_state =
CLOSED ) Agent: air_motor
----------
Cycle:12 State:
gas_valve_state=CLOSED
air_valve_state=CLOSED
igniter_state=OFF
switch_switch_state=OFF
flame_detector_flame_state=PRESENT
Active List:  G1, G3, G8, G16, G4, G22
Action List:
----------
```

Again, the system was forced into enter an unsafe state by opening the gas valve while the switch is OFF and the air valve is CLOSED; then, it recovered to a state where it closed the gas valve.

In figure 7.23, the sequence solution version animation data is represented using a STD. Similar to the STD in figure 7.21, the bold bordered rectangles represent the stable states while the others represent the transient states. Each stable state in figure 7.23 corresponds to one stable state in figure 7.21, states *St1*, *St4* and *St6* to states *St1*, *St5* and *St7*, respectively.

Although the two STDs look similar to each other, the sequential version possesses one advantage, which is that each of the transient states has a single operating terminal goal (in state *St11*). This leads to having a separate transient state for each terminal goal. This prunes the effort required to further investigate these transient states.

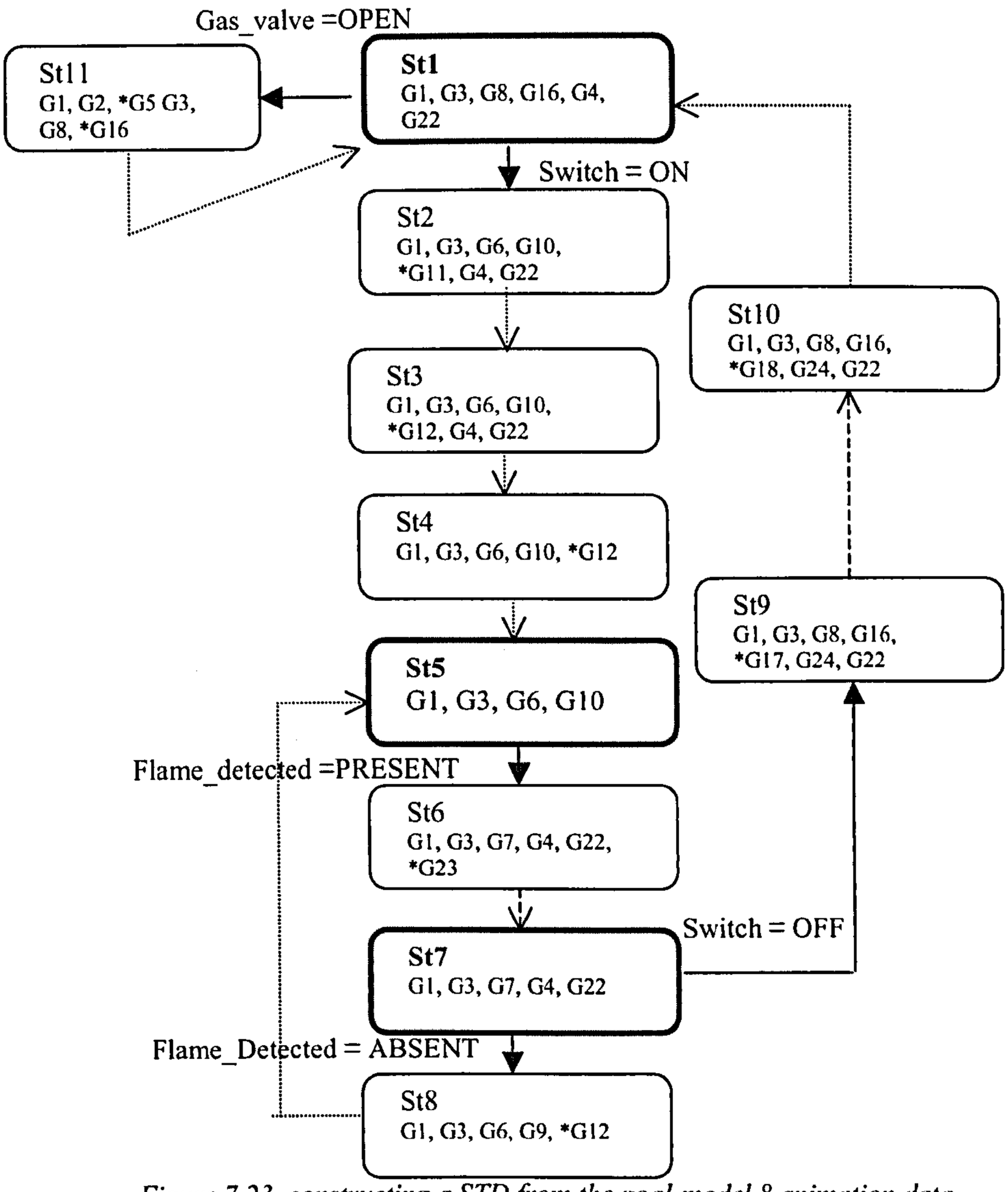


Figure 7.23, constructing a STD from the goal-model 8 animation data

7.4 Generating formal specifications

After validating the requirements, the tool can translate the goal-model created and accepted by the systems engineer to formal specification formats that can be processed by a software engineer. The tool documents the formal specifications as much as possible by integrating the informal description at the appropriate places with the formal description.

7.4.1 Generating formal operations/Invariants

Operational invariants can be generated from the terminal goals to be used generally, independently of a formal method or programming language. The tool generates operations with pre- and post-conditions and an actor or agent who is capable of performing such operation. Each operational terminal goal (neither environmental nor non-functional) will be translated into an invariant.

Table 7.6 the operations of goal-model 8

Invariant no. 1 closeGasValve Actor(Agent): gas_motor
/* G5: if the gas valve is open and air valve is closed,close gas valve*/ Pre-Condition: (gas_valve_state = OPEN) and air_valve_state=CLOSED and gas_valve_state= OPEN Post-Condition: gas_valve_state = CLOSED
Invariant no. 2 openAirValve Actor(Agent): air_motor
/* G11: Open the valve,*/ Pre-Condition: (air_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) Post-Condition: air_valve_state = OPEN
Invariant no. 3 openGasValve Actor(Agent): gas_motor
/* G12: Open the valve,*/ Pre-Condition: (gas_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) and air_valve_state = OPEN Post-Condition: gas_valve_state = OPEN
Invariant no. 4 switch_on Actor(Agent): igniter_ignitior
/* G13: to switch on the igniter to attempt to produce spark,*/ Pre-Condition: (igniter_state = OFF) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) and air_valve_state = OPEN and gas_valve_state = OPEN Post-Condition: igniter_state = ON
Invariant no. 5 openGasValve Actor(Agent): gas_motor
/* G14: Open the valve,*/ Pre-Condition: (gas_valve_state = CLOSED and air_valve_state=OPEN) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT) Post-Condition: gas_valve_state = OPEN
Invariant no. 6 openAirValve Actor(Agent): air_motor
/* G15: Open the valve,*/ Pre-Condition: (air_valve_state = CLOSED) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT) Post-Condition: air_valve_state = OPEN
Invariant no. 7 closeGasValve Actor(Agent): gas_motor
/* G17: to close the valve,*/ Pre-Condition: (gas_valve_state = OPEN) and (switch_switch_state = OFF) and (flame_detector_flame_state = PRESENT) Post-Condition: gas_valve_state = CLOSED
Invariant no. 8 closeAirValve Actor(Agent): air_motor
/* G18: to close the valve,*/ Pre-Condition: (air_valve_state = OPEN) and (switch_switch_state = OFF) and (flame_detector_flame_state = PRESENT) and gas_valve_state = CLOSED Post-Condition: air_valve_state = CLOSED
Invariant no. 9 closeAirValve Actor(Agent): air_motor
/* G19: to close the valve,*/ Pre-Condition: (air_valve_state = OPEN) and switch_switch_state=OFF and flame_detector_flame_state=ABSENT Post-Condition: air_valve_state = CLOSED
Invariant no. 10 closeGasValve Actor(Agent): gas_motor
/* G20: to close the valve,*/ Pre-Condition: (gas_valve_state = OPEN) and switch_switch_state=OFF and flame_detector_flame_state=ABSENT Post-Condition: gas_valve_state = CLOSED
Invariant no. 11 switch_off Actor(Agent): igniter_ignitior
/* G21: switch off the spark,switch off the spark ignitor*/ Pre-Condition: (igniter_state = ON) and switch_switch_state=OFF and flame_detector_flame_state=ABSENT Post-Condition: igniter_state = OFF

Invariant no. 12 switch_off Actor(Agent): igniter_ignitior
/* G23: switch off the spark,switch off the spark ignitor*/ Pre-Condition: (igniter_state = ON) and flame_detector_flame_state=PRESENT or (flame_detector_flame_state=ABSENT and (gas_valve_state=CLOSED or air_valve_state=CLOSED)) Post-Condition: igniter_state = OFF

The simultaneous version has fewer operations compared to the sequential version because some of the terminal goals achieve more than one function, as explained in the section where the incompleteness cases were removed by compacting the two goals to ensure closing and shutting down.

Table 7.7, the operations of goal-model 9

Invariant no. 1 closeGasValve Actor(Agent): gas_motor
/* G5: if the gas valve is open and air valve is closed,close gas valve*/ Pre-Condition: (gas_valve_state = OPEN) and air_valve_state=CLOSED and gas_valve_state= OPEN Post-Condition: gas_valve_state = CLOSED
Invariant no. 2 openAirValve Actor(Agent): air_motor
/* G10: Open the valve,*/ Pre-Condition: (air_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) Post-Condition: air_valve_state = OPEN
Invariant no. 3 openGasValve Actor(Agent): gas_motor
/* G11: Open the valve,*/ Pre-Condition: (gas_valve_state = CLOSED and air_valve_state=OPEN) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) Post-Condition: gas_valve_state = OPEN
Invariant no. 4 switch_on Actor(Agent): igniter_ignitior
/* G12: to switch on the igniter to attempt to produce spark,*/ Pre-Condition: (igniter_state = OFF) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON) Post-Condition: igniter_state = ON
Invariant no. 5 openGasValve Actor(Agent): igniter_ignitior
/* G13: Open the valve,*/ Pre-Condition: (gas_valve_state = CLOSED and air_valve_state=OPEN) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT) Post-Condition: gas_valve_state = OPEN
Invariant no. 6 openAirValve Actor(Agent): air_motor
/* G14: Open the valve,*/ Pre-Condition: (air_valve_state = CLOSED) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT) Post-Condition: air_valve_state = OPEN
Invariant no. 7 closeAirValve Actor(Agent): air_motor
/* G15: to close the valve,*/ Pre-Condition: (air_valve_state = OPEN) and switch_switch_state=OFF Post-Condition: air_valve_state = CLOSED
Invariant no. 8 closeGasValve Actor(Agent): gas_motor
/* G16: to close the valve,*/ Pre-Condition: (gas_valve_state = OPEN) and switch_switch_state=OFF Post-Condition: gas_valve_state = CLOSED
Invariant no. 9 switch_off Actor(Agent): igniter_ignitior
/* G17: switch off the spark,switch off the spark ignitor*/ Pre-Condition: (igniter_state = ON) and switch_switch_state=OFF Post-Condition: igniter_state = OFF

Invariant no. 10 switch_off Actor(Agent): igniter_ignitior
/* G19: switch off the spark,switch off the spark ignitor*/ Pre-Condition: (igniter_state = ON) and flame_detector_flame_state=PRESENT and switch_switch_state= ON Post-Condition: igniter_state = OFF

7.4.2 Generating B machines

The GOPCSD tool translates the goal-model automatically to a specification form as B machines. In this case study, there are three agents controlling the two valves and the igniter, and each of them has two basic operations represented as the component's low-level goal-models 1, 2, 3, 4, 5, and 6.

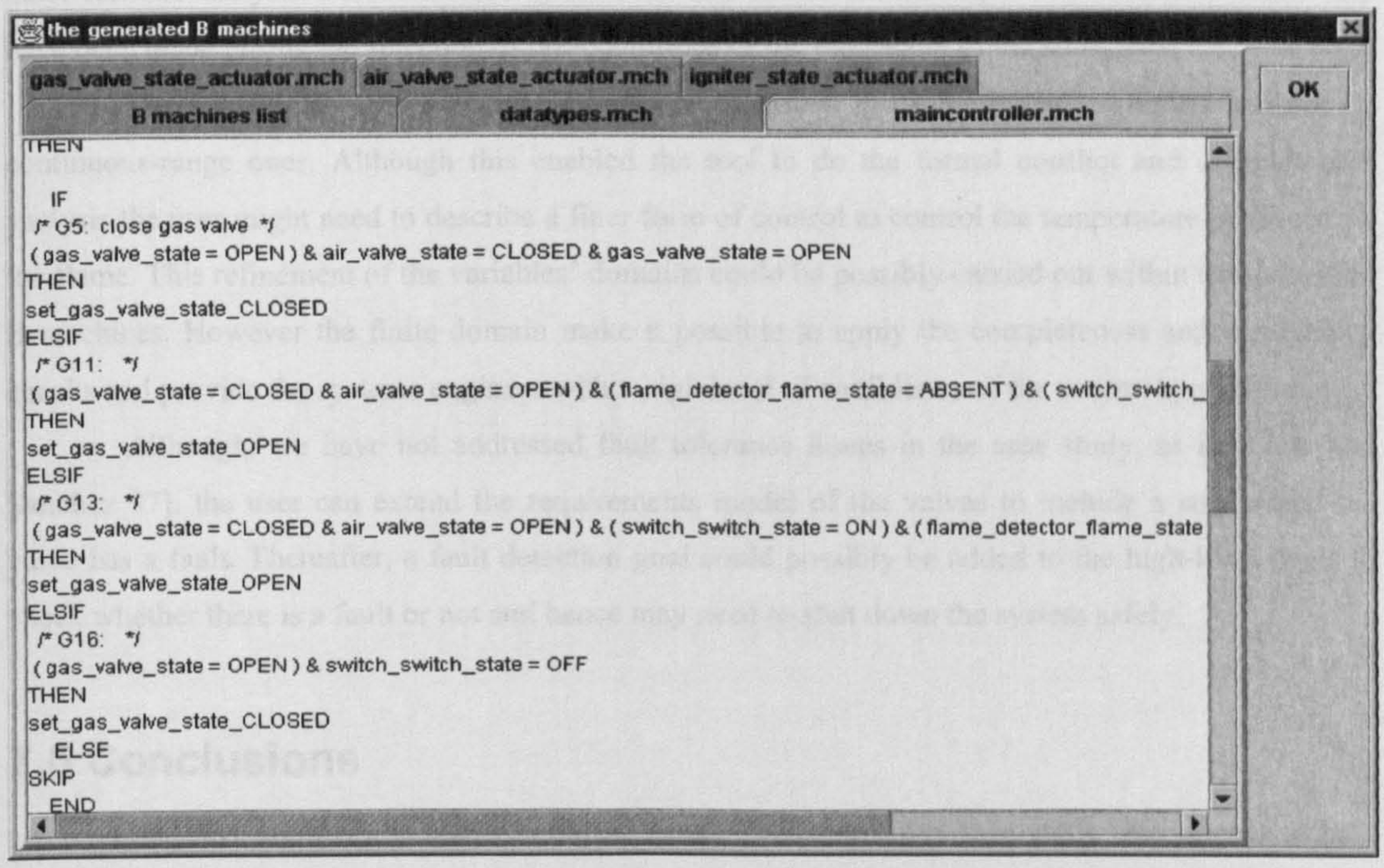


Figure 7.24, the generated B machines

For each version of the sequential and simultaneous goal-models, the tool will generate a complete set of B machines to control the gas burner. The generated machines have three types as follows:

- A Data Type machine that stores the definition of the enumerated data types used in the other machines.
- Actuators machines representing each agent and the operations that the agent can perform.
- A main controller machine that represents the entire goal-model, it includes the actuator machines. The controller machine has operations representing the agents' contribution to the entire goal-model; each of these operations will be an IF_THEN_ELSE structure of the terminal goals grouped by their assigned agents and, instead of changing the output variables directly, the appropriate operation from one of the actuator machines will be invoked.

Figure 7.24 shows a segment of the main controller machine that represents goal-model 9. The upper tabs of the dialogue box show the names of the different generated machines in addition to an index file "B machines list" that lists the B machines.

A full list of the generated B-machines for the simultaneous and sequential versions is provided in appendix B, section B.1.3.

7.5 Discussion

The tool enabled the user to start from abstract informal requirements and then gradually increasing the detail and formality until reaching a stage where the requirements are formally specified and structured in a single goal-model. Although the tool enabled the user to develop two different versions for the gas burner application at the same time, it did not provide guidance for preferring or evaluating each separate solution. The user has to make this choice in some other way and using some other form of analysis. Furthermore, it guided the user to correct the imperfect decisions, which may have been made during the selection of pre-conditions or the placement of the sub-goals.

The air and gas valves and the switch are described by finite domain variables and not by continuous-range ones. Although this enabled the tool to do the formal conflict and completeness analysis the user might need to describe a finer form of control to control the temperature produced by the flame. This refinement of the variables' domains could be possibly carried out within the generated B machines. However the finite domain make it possible to apply the completeness and consistency checks and provide the systems engineer with a high level of confidence of the control application.

Although, we have not addressed fault tolerance issues in the case study, as in [Lano and Sanchez 97], the user can extend the requirements model of the valves to include a state when the valve has a fault. Thereafter, a fault detection goal could possibly be added to the high-level goals to check whether there is a fault or not and hence may need to shut down the system safely.

7.6 Conclusions

We have studied a simple gas burner system. Although this case study may appear to be a small one, it demonstrated some promising aspects of the GOPCSD tool; the tool does not restrict the user to begin the controller development lifecycle with a perfect design. It guides its user to reach this stage after a feedback loop of checking and modifying the requirements. The user can approach the design problem differently as seen with the two solution versions; then, the tool guides him/her to modify the requirement specifications.

The developed application can be extended within the B-toolkit environment by adding programmatic refinement. In addition, the application may be reused, as a sub-system in a higher system by using the entire goal-model.

Case Study II
The Production Cell

8

In this chapter, we present a production cell case study to assess the tool (and hence the method of GOPCSD) when dealing with a medium-scale application, such as the production cell. The liveness and safety aspects of the production cell are considered as well as correctness and throughput. We indicate how the application requirements design problem can be divided into small segments to reduce the effort. Then, we explore the possibilities of extending the resulting goal-oriented requirements of the case study corresponding to enhanced requirements. Finally, we comment about the requirements structure aspects.

8.1 Introduction

Our second case study is a general production cell [Lewerentz and Lindner 95]; this production cell is considered as a medium scale application and it has been studied in the past using different specification analyses [Lewerentz and Lindner 95, Zorzo et al. 99, Piveropoulos Y2K, Winter 01].

Production cells are a wide family of process control systems in which there is a production line that processes a product or a set of products in a pipeline arrangement. For instance, production cells can be utilised to integrate electronic boards, stamp blank metals, and paint cars. Although there is a considerable variation in the purpose of employing the specific production cells, similar components can be found within these different applications, such as robots, feed belts and deposit belts. Furthermore, many similar high-level goals can be generally formulated for production cells independently of the particular applications.

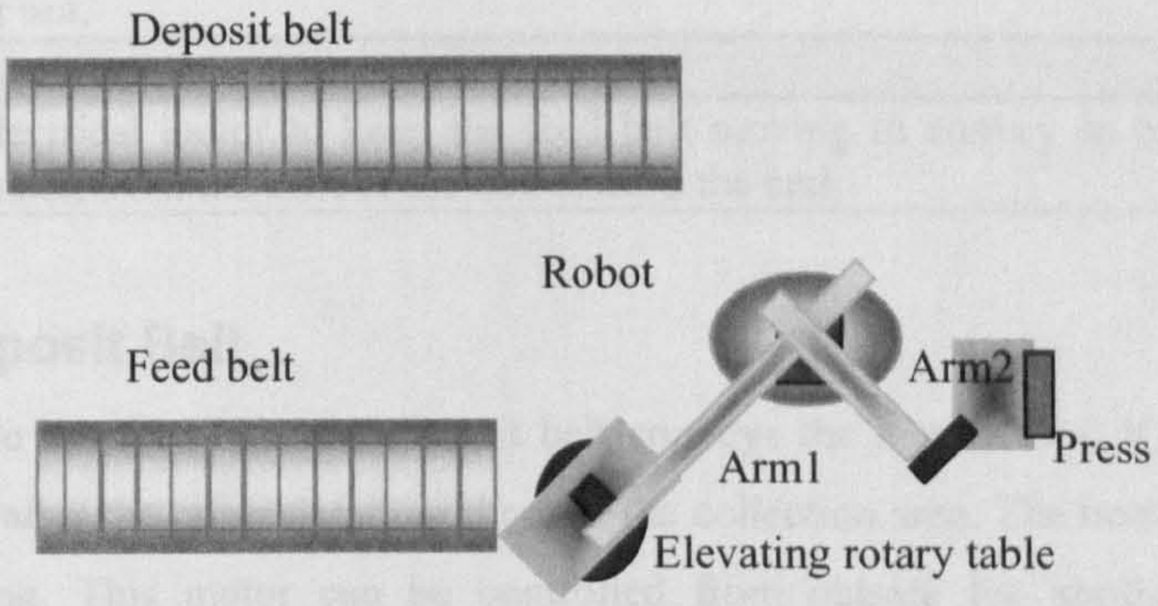


Figure 8.1, The Production cell

Figure 8.1 shows a simple production cell consisting of a feed belt, a deposit belt, an elevating rotary table, a two-arm robot and a press (the processing unit). This production cell processes blank metals conveyed through the feed belt to the elevation rotary table. The robot picks the elements from the table, and places them inside the press until they get pressed. Finally, it moves them to the deposit belt that conveys the processed metals to the collection area or further processing stages.

In the following sub-sections we provide a brief description of these components as a basic understanding of production cells in general. However, different production cells can have different arrangement and different processing units; for example, in car painting production lines, various robot models are employed to paint the car rather than moving the products.

8.1.1 The Feed Belt

The feed belt is constructed as shown in figure 8.2. It is controlled by an on/off motor, which keeps the metals moving in one direction. The belt has two photocells (S1, S2) to sense the arrival of the metals at the start or end parts of the belt. In table 8.1, we briefly list the sensors and the actuators of the feed belt.

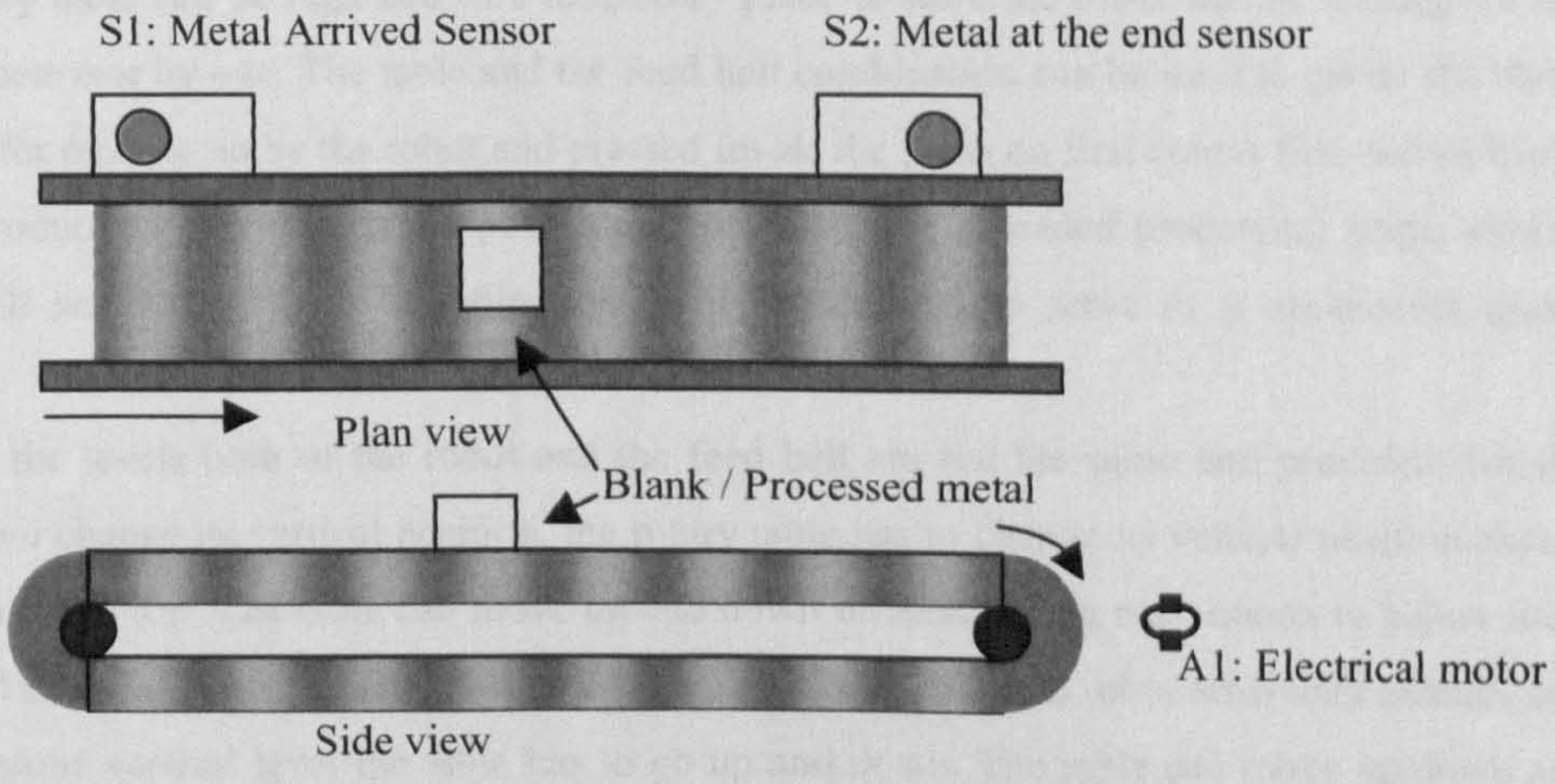


Figure 8.2, the feed belt component

Table 8.1, sensors and actuators of the feed belt

No.	***Name*, Description of the sensor**	**Type**
S1	*metal_at_start*, A photocell to indicate whether there is a metal at the start of the feed belt or not.	Yes/No
S2	*metal_at_end*, A photocell to indicate whether there is a metal at the end of the	Yes/No

	feed belt or not.	
No.	***Name*, Description of the actuator**	**Type**
A1	*moving*, Electrical motor to keep the feed belt moving to convey in order to move the metal from the start of the belt toward the end.	ON /OFF

8.1.2 The Deposit Belt

Similar to the feed belt, the deposit belt conveys the metals. But, it delivers the processed (stamped) metals after the press stamping them to the collection area. The belt has a motor that keeps the metals moving. This motor can be controlled from outside the application and not by the production cell software controller because it is dependent on the next stages or the collection status. For example, if the metals are collected in boxes there may be some manual or automatic interaction from time to time to move the full boxes and provide empty ones. Like the feed belt, the deposit belt has two photocell sensors S1 and S2, to indicate the arrival of processed metals at the start and the end parts of the belt, respectively. Table 8.2 lists the sensors and actuators of the deposit belt:

Table 8.2, sensors and actuators of the feed belt

No.	***Name*, Description of the sensor**	**Type**
S1	*metal_at_start*, A photocell to indicate whether there is a metal at the start of the deposit belt or not.	Yes/No
S2	*metal_at_end*, A photocell to indicate whether there is a metal at the end of the deposit belt or not.	Yes/No
No.	***Name*, Description of the actuator**	**Type**
A1	*motor*, Electrical motor to keep the deposit belt moving to convey in order to move the metal from the start of the belt toward the end.	ON /OFF

8.1.3 The Rotary Table

The rotary table is used to raise the blank metals to the robot arm1 level so that it can pick them. The rotary table can be regarded as a temporary place to store the blank metals waiting for the robot to pick them one by one. The table and the feed belt combination can be used to queue the blank metals to wait for picking up by the robot and pressed inside the press on first comes first served basis; Thus, if the production cell application possesses more than one cascaded processing stage, similar extra "feed belt and rotary table" combinations will be required to serve as a multi-level queue (pipeline).

Since the levels both of the robot and the feed belt are not the same and provided that the robot arm cannot change its vertical position, the rotary table has to change its vertical position once a metal has been on its top. The table can move up and down or rotate using two motors to adjust itself to the level and angle of either the feed belt or the robot. Since each of the robot arms only extends and retracts at the same vertical level the table has to go up and down. The table can move up/down and rotate either simultaneously or sequentially depending on the obstruction of the robot's arms and feed belt. The throughput of the entire system can be improved by reducing the time required for the table to reach the feed belt to receive a new blank metal carry or to reach the robot to enable it picking up the metal. Table 8.3 lists the rotary table sensors and actuators.

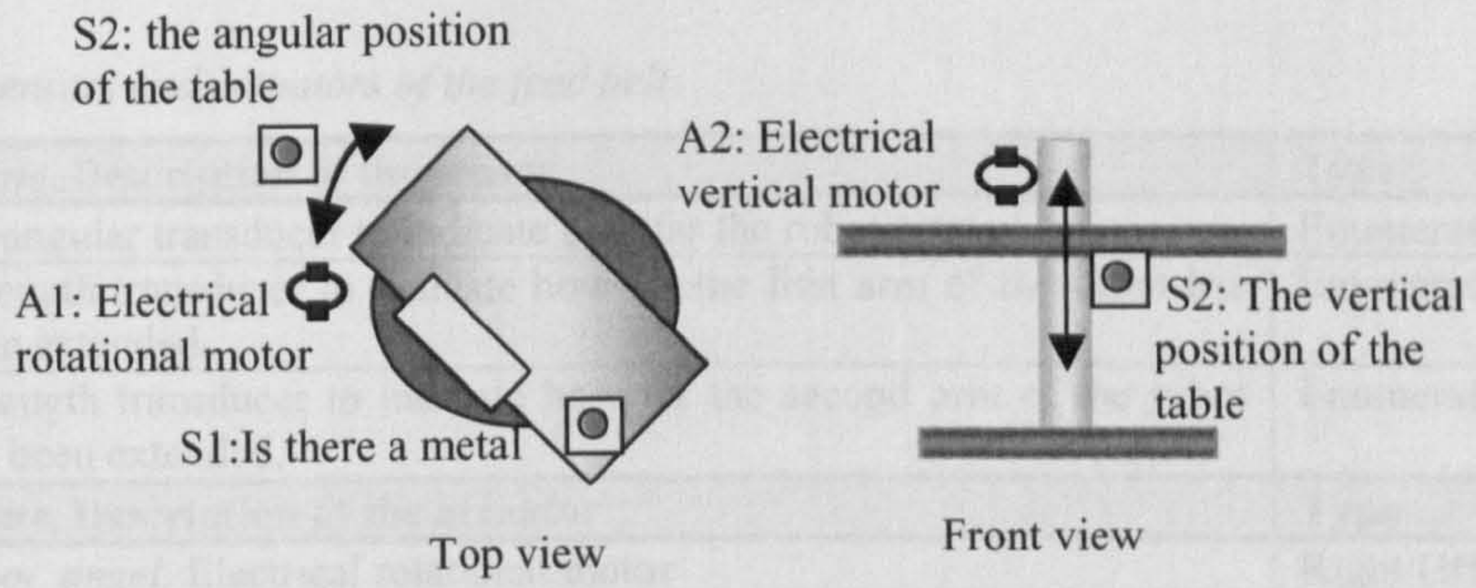


Figure 8.3, the elevation rotary table component

Table 8.3, sensors and actuators of the feed belt

No.	***Name*, Description of the sensor**	**Type**
S1	*table_has_metal,* Is there a metal	Yes/No
S2	*table_angle*, The angular position of the table	Enumerated
S3	*table_level,* The vertical position of the table	Up/ Down
No.	***Name*, Description of the actuator**	**Type**
A1	*rotational_motor*, Electrical rotational motor	Left/OFF/Right
A2	*vertical_motor,* Electrical vertical motor	Up/Off/Down

8.1.4 The Robot

Robots are important units of the production cell. They usually accomplish the tasks that human cannot do because of the dangerous environmental conditions like painting cars or holding very hot or cold products or when precision is highly required like welding and placing integrated circuits on electronic boards. The robot in this production cell has two responsibilities: first, moving the blank metals from the table to the press and second moving the stamped metals from the press to the deposit belt.

As shown in figure 8.4, the robot has a twistable body that is controlled by rotational motor to enable it to rotate towards the table, the press, and the start of the deposit belt. Further, the robot has two arms to pick up the metals from the table and place them inside the press and then, pick up them from the press and moving them to the deposit belt. The two arms can separately extend or retract. Each of them ends with a magnet, which can be energised and de-energised to enable picking and dropping the metals. Table 8.4 lists the related sensors and the actuators of the robot.

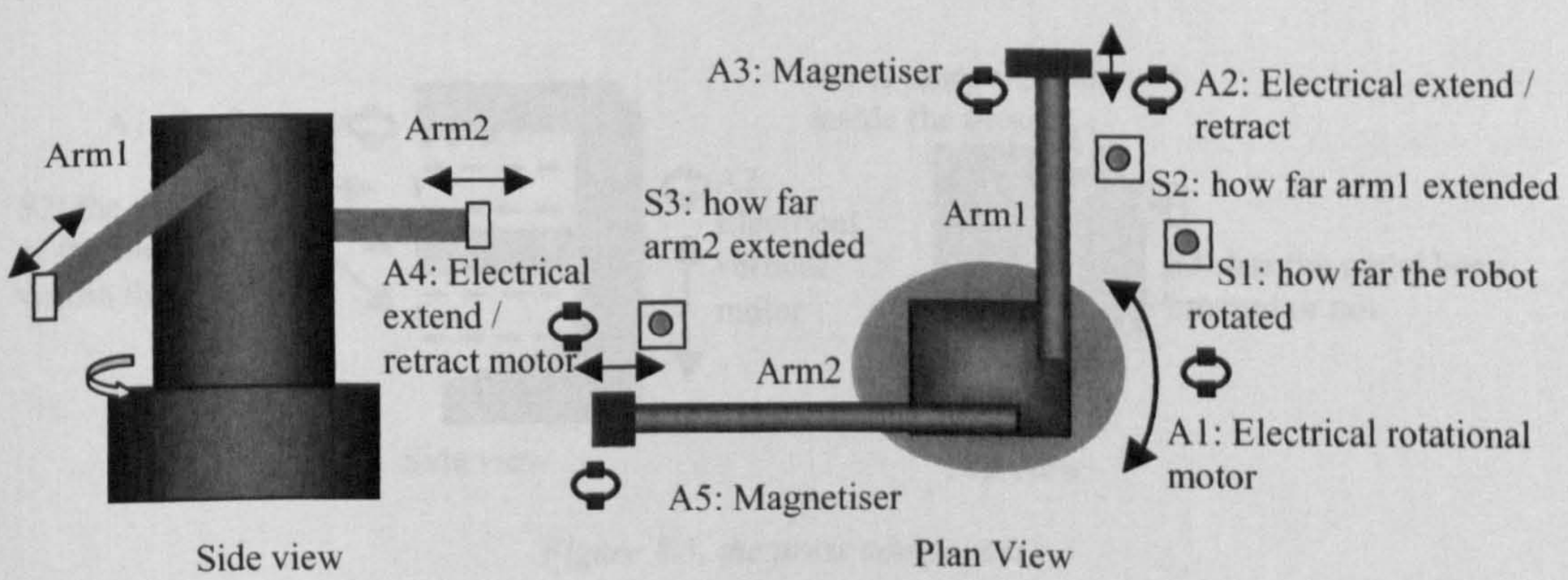


Figure 8.4, the Robot component

Table 8.4, sensors and actuators of the feed belt

No.	***Name*, Description of the sensor**	**Type**
S1	An angular transducer to indicate how far the robot rotated.	Enumerated range
S2	A length transducer to indicate how far the first arm of the robot has been extended.	Enumerated range
S3	A length transducer to indicate how far the second arm of the robot has been extended.	Enumerated range
No.	***Name*, Description of the actuator**	**Type**
A1	*robot_angel*, Electrical rotational motor	Right/Off/Left
A2	*arm1_length*, Electrical arm1 extend / retract	Extend/STOP/Retract
A3	*arm1_magnet*, Magnetiser arm1	ON/OFF
A4	*arm2_length*, Electrical arm2 extend / retract motor	Extend/STOP/Retract
A5	*arm2_magnet*, Magnetiser arm2	ON/OFF

8.1.5 The Press

The press is an example of a processor that accepts unprocessed input and then processes it. For example, the press stamps the manufacturing details on each blank metal. The robot places the blank metals inside the press; and after processing, it picks the processed metals to the deposit belt.

The press component has sensors to indicate whether there is a metal inside it or not and whether the metal has been pressed yet or not. It has two actuators: a motor that provides vertical motion to move the metal between three levels (upper for stamping, lower for picking by the robot arm2 and middle for receiving the new blank metals by robot arm1). Table 8.5 lists the sensors and the actuators of the press.

Table 8.5, sensors and actuators of the feed belt

No.	***Name*, Description of the sensor**	**Type**
S1	*Press_has_metal*, A weight sensor to indicate whether there is a metal inside the press or not.	YesNo
S2	*tray_level*, A position sensor to indicate the position of the metal inside the press.	UP/MIDDLE/DOWN
S3	*Metal_pressed a sensor or switch set by the presser after pressing each blank metal*	YesNo
No.	***Name*, Description of the actuator**	**Type**
A1	*pressing*, Presser to stamp the metals when they are on the upper position inside the press.	ON /OFF
A2	*vertical_motor*, Electrical vertical motor to move the tray that carries the metal inside the press	UP/STOP/DOWN

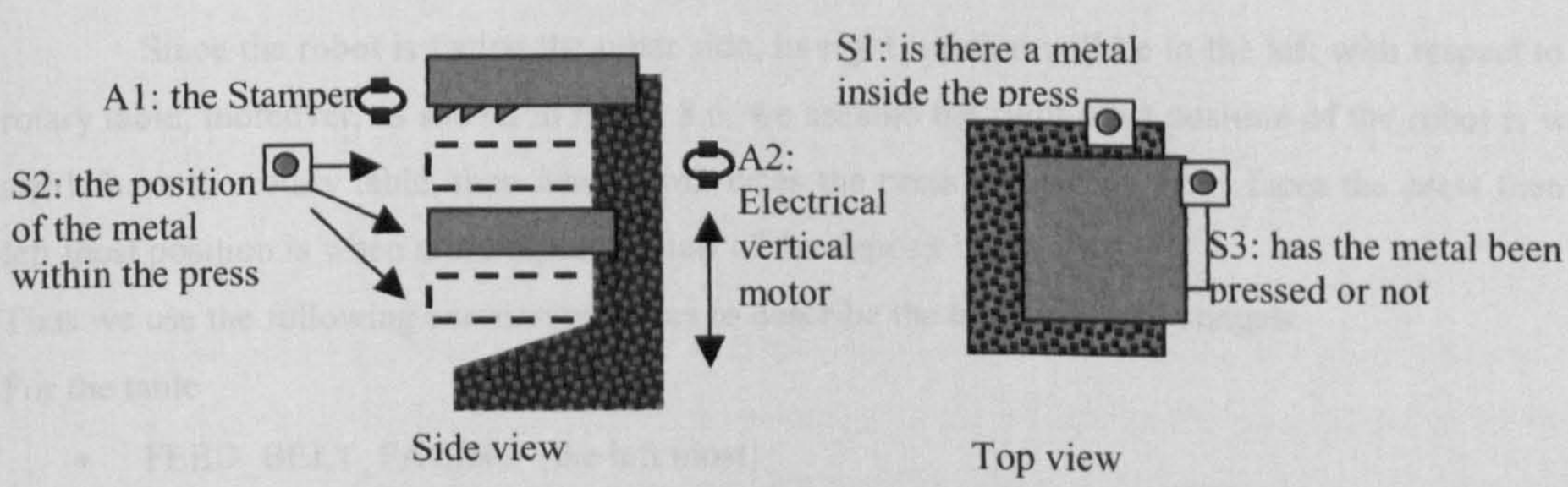


Figure 8.5, the press component

8.1.6 The Operation of the production cell

Figure 8.6 shows the different final positions to which the robot can be rotated. As shown, one of the robot arms is retracted, and the other one is extended to pick up or drop the metal, while the other one should be retracted. This should be maintained because of the fact that keeping the arms retracted while the robot rotates should decrease the chances of hitting any nearby objects. The robot can be set to rotate to different angles as shown in the figure. In our treatment we will assume the following:

The elevation rotary table rotates to the right to move from the position where it faces the feed belt to the position where it faces the robot, and to left vice versa, taking the short journey.

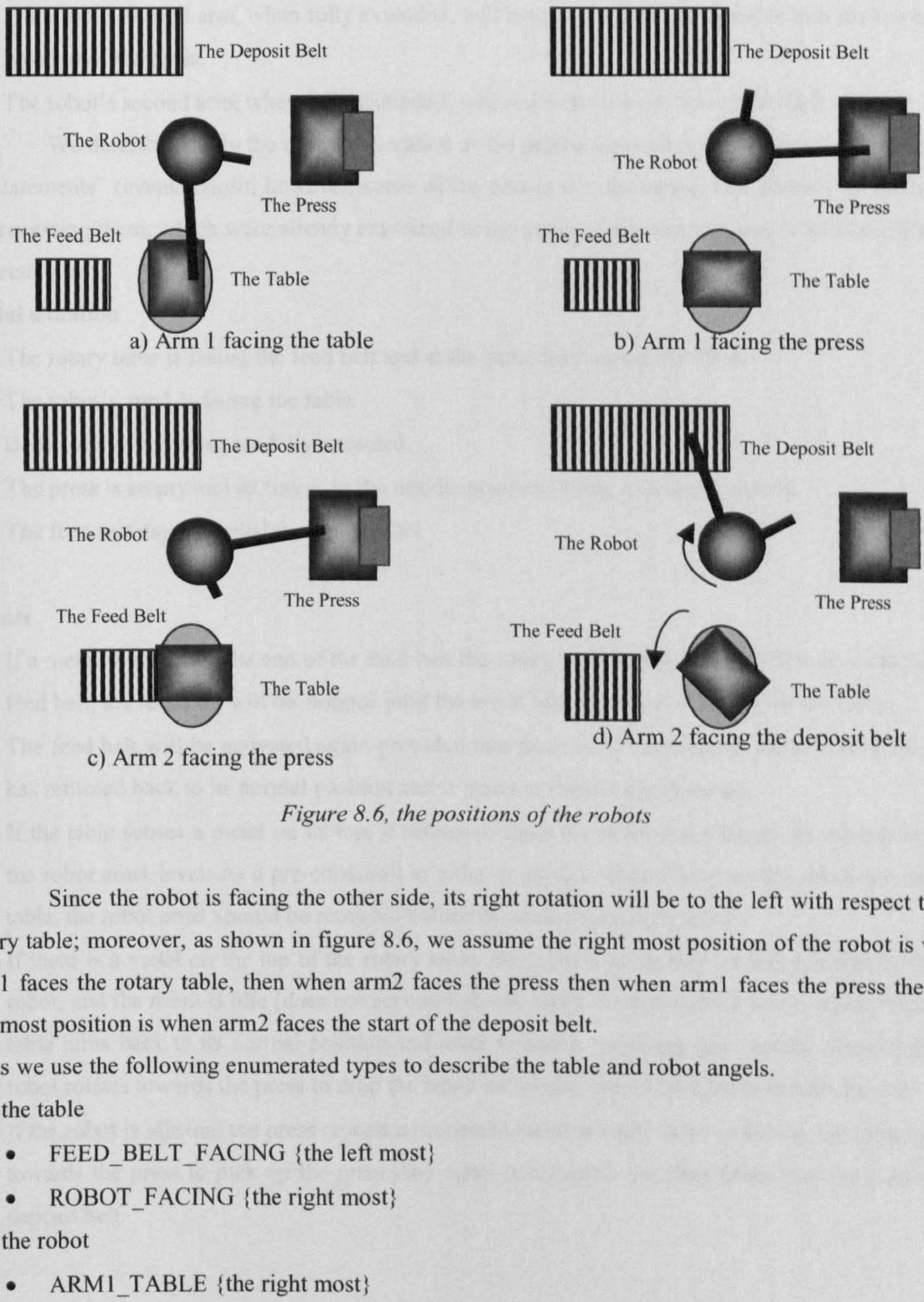


Figure 8.6, the positions of the robots

Since the robot is facing the other side, its right rotation will be to the left with respect to the rotary table; moreover, as shown in figure 8.6, we assume the right most position of the robot is when arm1 faces the rotary table, then when arm2 faces the press then when arm1 faces the press then the left most position is when arm2 faces the start of the deposit belt.

Thus we use the following enumerated types to describe the table and robot angels.

For the table

- FEED_BELT_FACING {the left most}
- ROBOT_FACING {the right most}

For the robot

- ARM1_TABLE {the right most}

- ARM2_PRESS
- ARM1_PRESS
- ARM2_DEPOSIT_BELT {the left most}

The different components of the production cell application should be placed in a configuration that satisfies the following constraints:

- The robot's first arm is at the same level as the upper position of the rotary table.
- The robot's first arm, when fully extended, reaches the blank metals on the rotary table.
- The robot's second arm, the deposit belt and the lower position of the press tray have the same vertical level.
- The robot's second arm, when fully extended, will reach the processed metal within the lower tray position of the press.
- The robot's second arm, when fully extended, will reach the start of the deposit belt.

We describe briefly the normal operation of the production cell as initial situation and a set of "if statements" (event/action); however, some of the details are abstracted, like picking up the metals or processing them, which were already explained in the various previous sections or will be explained in detail later:

Initial situation

- The rotary table is facing the feed belt and at the same level as the feed belt.
- The robot's arm1 is facing the table.
- Both arms of the robot are fully retracted.
- The press is empty and its tray is in the middle position (ready to stamp a metal).
- The feed and deposit belts' motors are ON.

Events

- If a metal is sensed at the end of the feed belt the rotary table is checked; and if it is not facing the feed belt, the feed belt will be stopped until the metal will be placed at the top of the table.
- The feed belt will be activated again provided that there is no blank metal at the end or the table has returned back to its normal position and is ready to receive blank metals.
- If the table senses a metal on its top, it rotates towards the robot and changes its vertical level to the robot arm1 level. As a pre-condition in order to avoid collision between the robot arm and the table, the robot arm1 should be retracted before the table changes its level.
- If there is a metal on the top of the rotary table, the table is at the top vertical position facing the robot, and the robot is idle (does not accomplish any task), the robot picks up the metal. Then, the table turns back to its normal position and level to enable receiving new metals. Meanwhile, the robot rotates towards the press to drop the blank metal and moves back to its normal location.
- If the robot is idle and the press reports a processed metal is ready to be collected, the robot rotates towards the press to pick up the processed metal using arm2 and then drops it at the start of the deposit belt.

Some issues should be noticed as follows:

- The production cell operations can be regarded as a pipeline of the following stages: conveying the metal to the table (feed belt responsibility), enabling the metal to be picked up by the robot (table responsibility), moving the metals toward the press by the robot, processing the metals in the press, moving the processed metals to the deposit belt by robot and finally, conveying the metal to the collection area by the deposit belt.
- Some of these pipeline stages allow more than one metal to be served at the same time, such as conveying metals on belts, but the other stages, like the robot arms, table and press, restrict the number of served metals to only one because of their physical nature.
- If there is situation in which there is a conjunction of a metal on the table ready for picking and a processed metal inside the press ready for delivering, the robot should empty the press first. Possibly, before moving the blank metals, the application should check whether the press is ready or not.

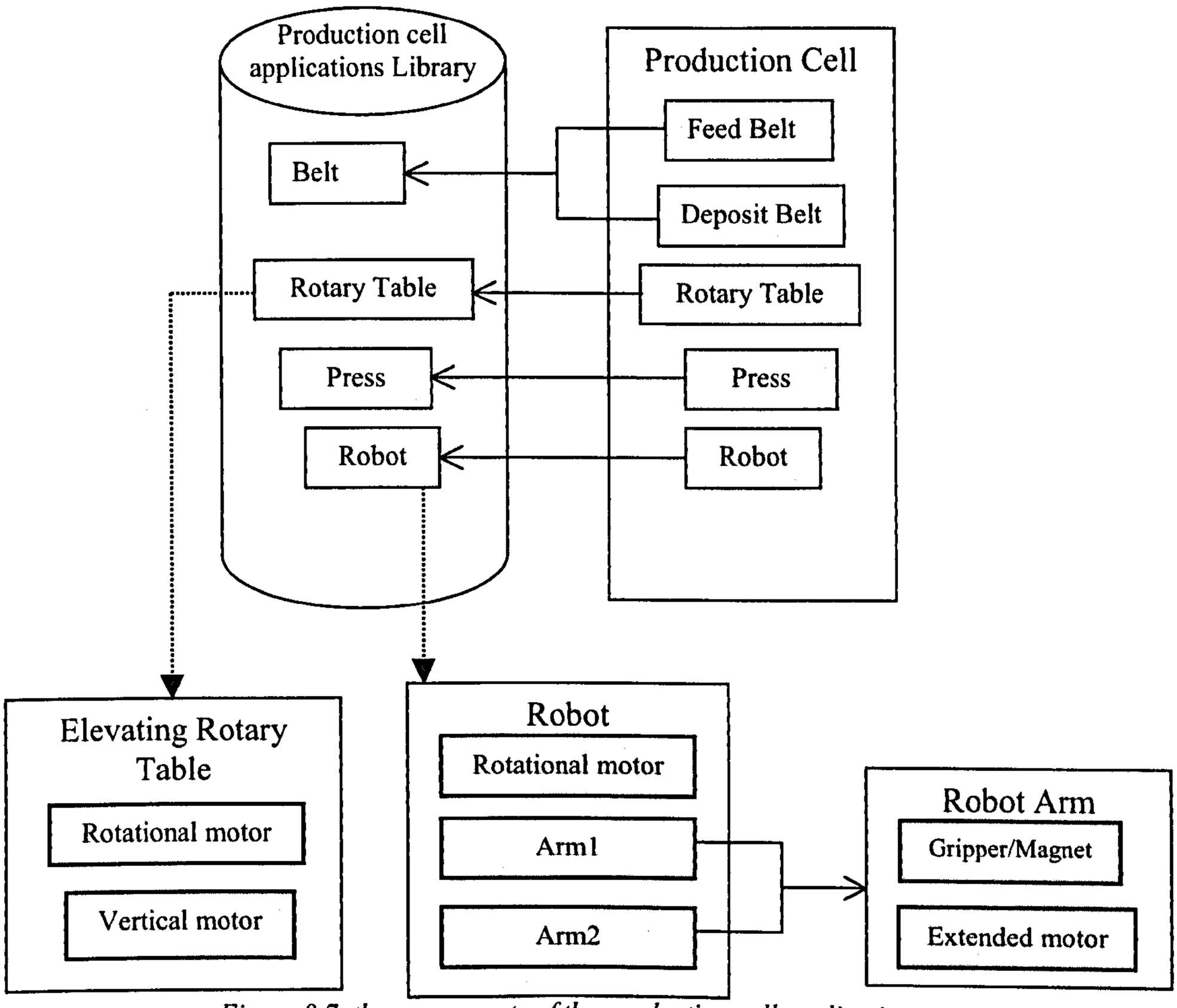


Figure 8.7, the components of the production cell application

8.2 Constructing the requirements

The construction phase is the first development phase, where the systems engineer will be guided to construct final versions of a goal-model. These developed goal-models consider the other aspects of the production cell apart from operation, like safety, liveness and throughput.

The user creates a new application and describes the function and the construction of the production cell.

8.2.1 Importing the Production Cell Components

The systems engineer should start the development of the production cell by identifying the physical components, like the feed belt, deposit belt, robot, press and rotary table; then, he/she can import them from the provided production cell library, using the import/component sub-menu, as shown in figure 8.7.

It should be noted that, unlike the gas burner case study in chapter 7, the separate components of the production cell have more compound low-level goal-models, which arise from the fact that they have their own sensor and actuator interactions and internal cycles of operations. Thus, the need for storing the components specifications will increase in this case, as well as the need for reducing the development effort.

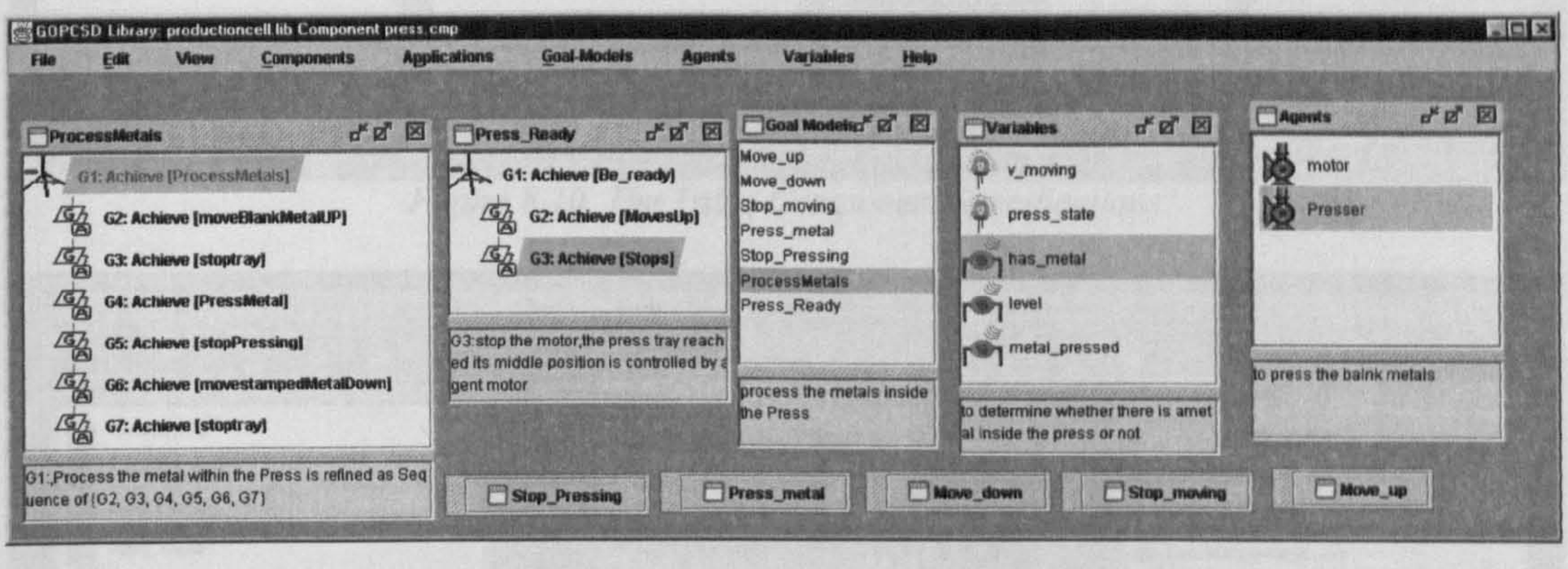


Figure 8.8, the Press Component Specifications

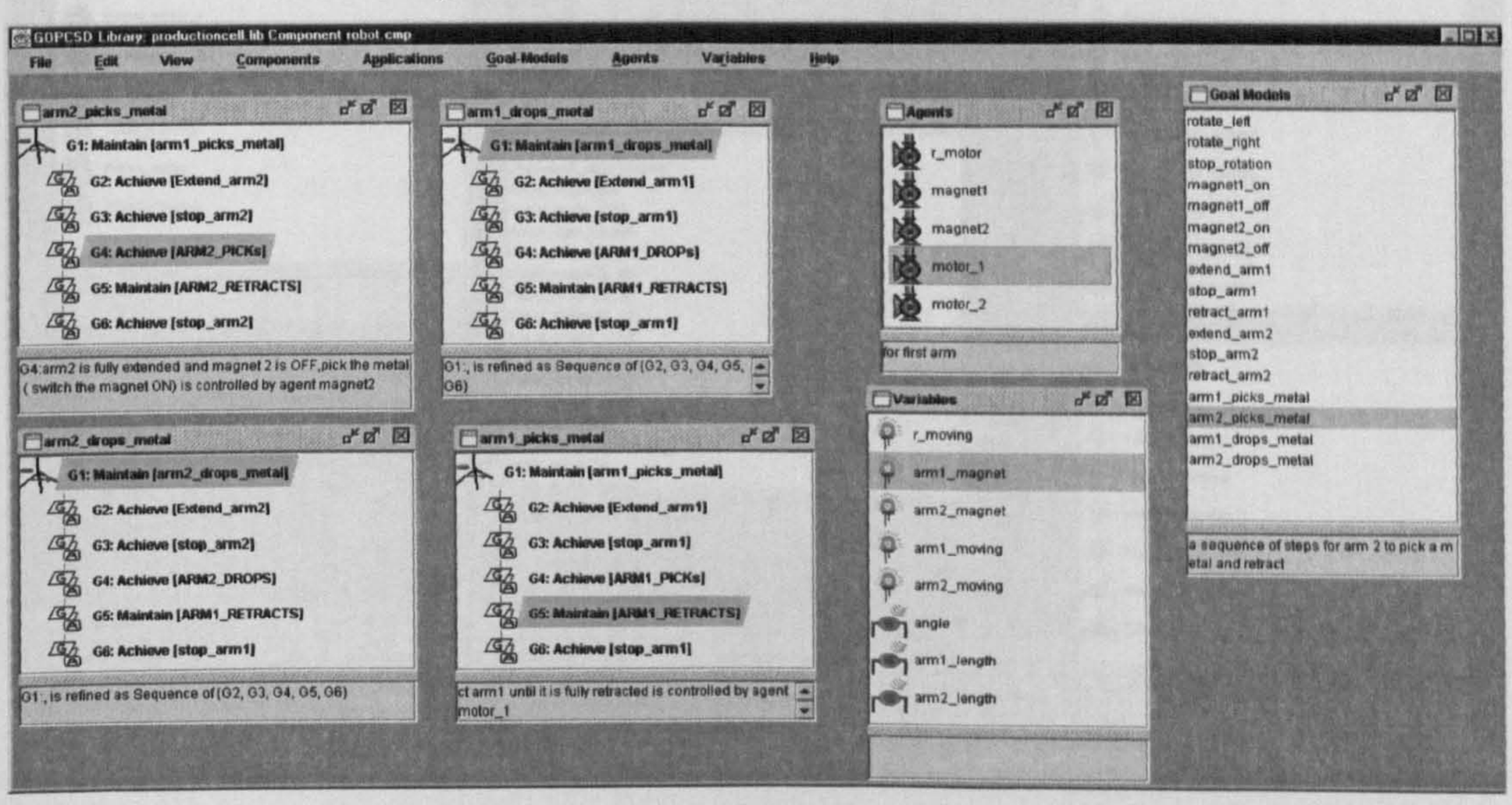


Figure 8.9, the Robot Component Specifications

Figure 8.8 shows the GOPCSD Library Manager desktop displaying the Press component. The *ProcessMetal* low-level goal-model specifies the lifecycle of pressing a blank metal, and in shown to the left as a sequence of six sub-goals, which move the blank metal to the upper position, stamp it, and move the processed metal to the lower position waiting for the robot to pick it.

Similarly, figure 8.9 shows the details of the robot component; the robot has requirements specifications for both the basic operations each of its agents can perform, and the compound operations performed as sequence of simple operations of the robot's magnets and the two arms, such as *arm1_picks_metals, arm1_drops_metals, arm2_picks_metals, and arm2_drops_metals.* Each of these compound goals is refined to simple operations; the entire compound goal can be copied to the application, to shorten the development time. The appropriate pre-conditions can then be added to the copied instances of these goal-models within the application.

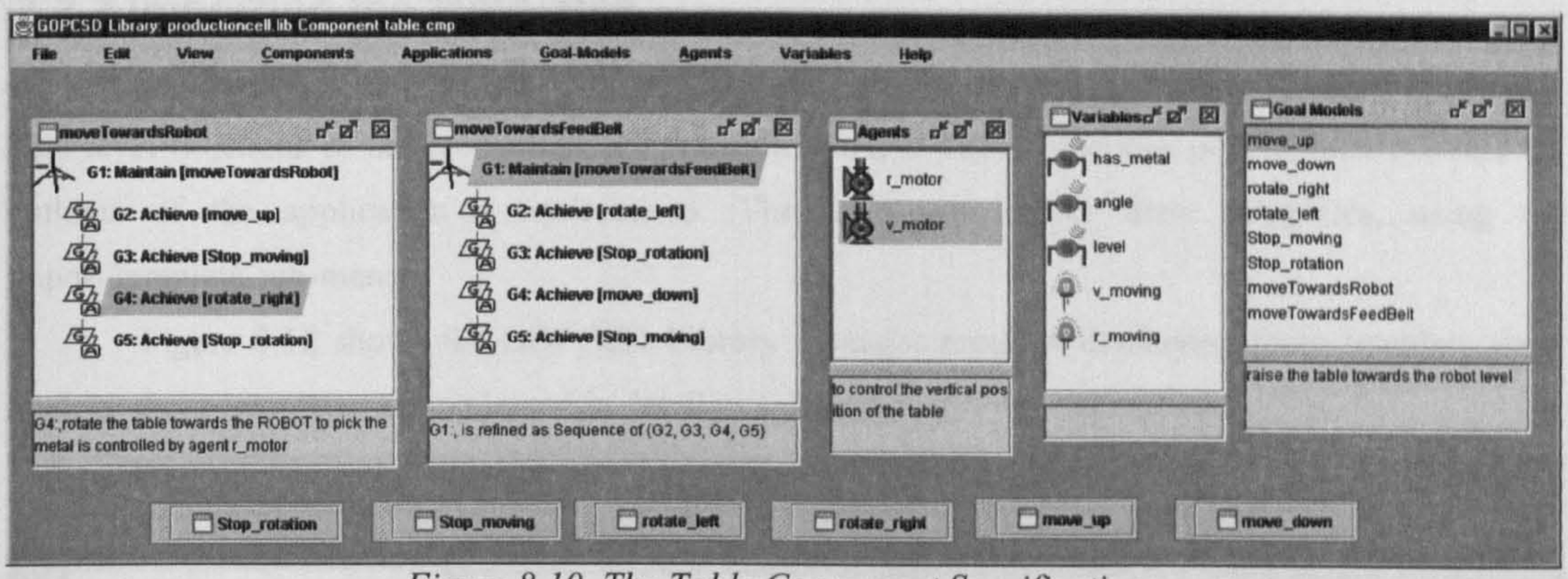


Figure 8.10, The Table Component Specifications

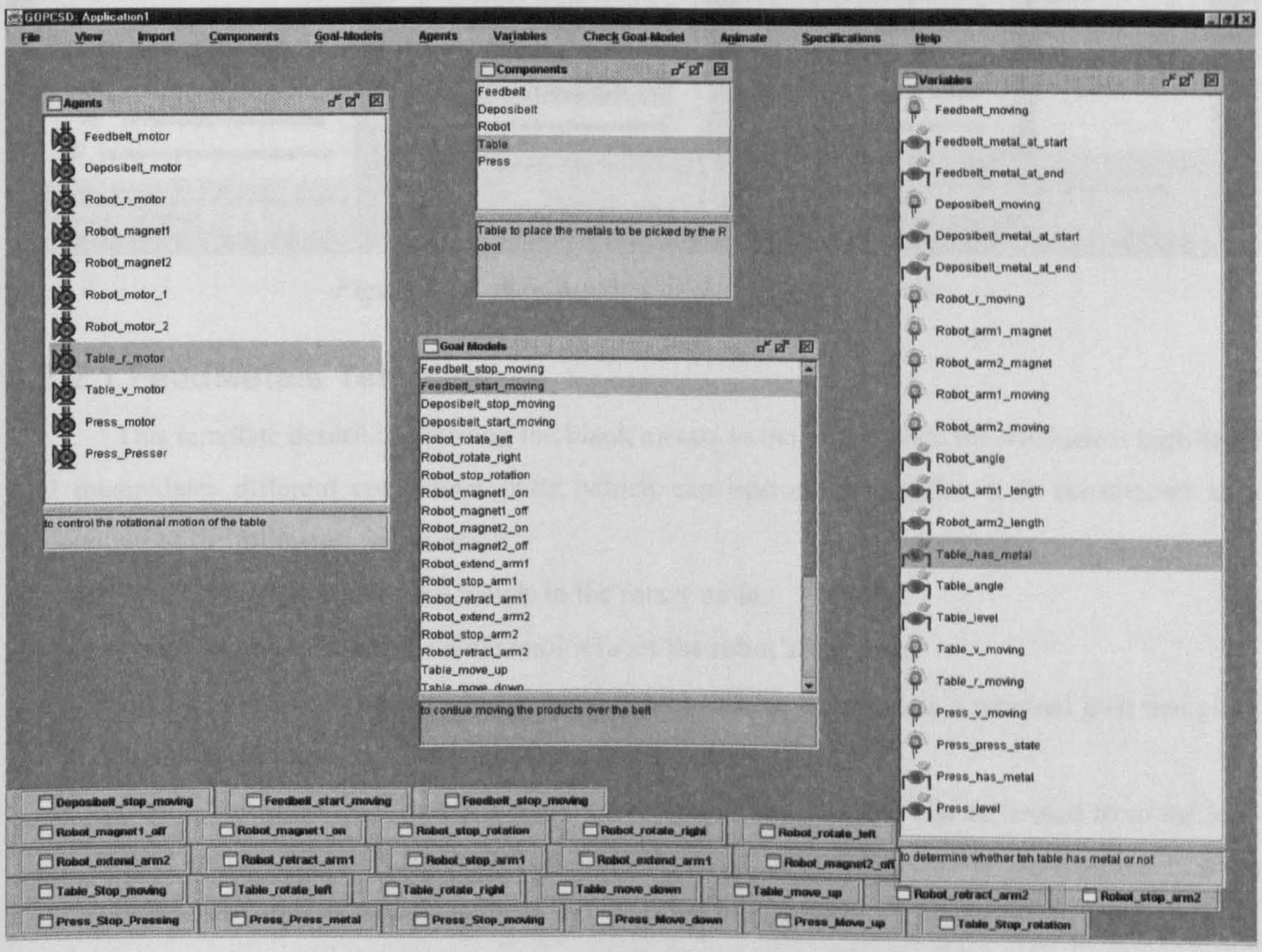


Figure 8.11, the GOPCSD tool desktop after importing the components

Similarly, as shown in figure 8.10, the Table component has two compound low-level goal-models to specify how the table can change its vertical level and angel to travel from the position facing the feed belt to the position facing the Robot, and vice versa. Each of these two compound goals is refined as a sequence of simple goals, as shown in the figure.

Because the components of the production cell do not share any sensors or actuators, it is better to copy from the library the entire sets of variables and agents of each component using the component name as a prefix. This conventional naming should make the requirements more readable and guide the user during the development, especially in such a medium sized application.

After importing the components, the application will look similar to the GOPCSD tool desktop shown in figure 8.11. The details of the components are listed in appendix B, B.2.1.

8.2.2 Importing the templates

In addition to the imported components, the systems engineer probably needs to import the high-level functions of the production cell from the library. These functions or templates provide the outlines of the application's requirements. The user can import these templates, using the import/template sub-menu.

Figure 8.12 shows the GOPCSD Library manager program displaying three template goal-models; these templates are explained in detail as follows:

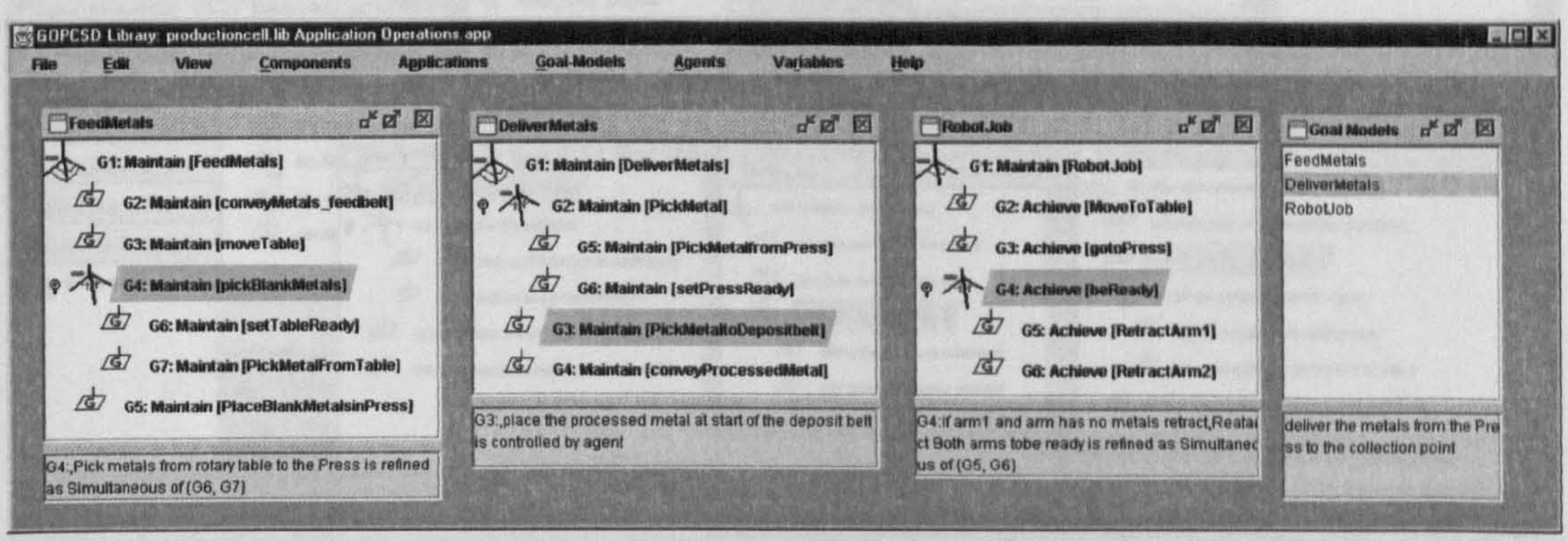


Figure 8.12, the templates of the Production Cell

8.2.2.1 FeedMetals Template

This template describing feeding the blank metals to the press; since the *feedmetals* high-level goal manipulates different component parts, which can operate in parallel, it is constructed as a conjunction of the following four goals:

- Goal GM36G2, Convey the metals to the rotary table.
- Goal GM36G3, Rotate the table until it faces the robot's first arm.
- Goal GM36G4, Pick up the blank metal, which can be specified as a terminal goal that picks up the blank metal on the rotary table using the robot's first arm.
- Goal GM36G5, Place the blank metal inside the press; this goal can be copied from the low-level goal model of the robot for arm drops the metal; after adding some guarding condition to ensure that arm1 is facing the press and the press has no metal.

8.2.2.2 DeliverMetals Template

This template involves delivering the processed metals from the press to the collection area; similar to the *feedmetals* goal, the *delivermetals* high-level goal is constructed as a conjunction of the following three goals:

- Goal GM37G2, Pick up the processed metals, which is can be specified as a terminal goal that picks up the processed metal from the press.
- Goal GM37G3, Place up the processed metals at the start of the deposit belt.
- Goal GM37G4, Convey the processed metal to the collection area (end of the deposit belt).

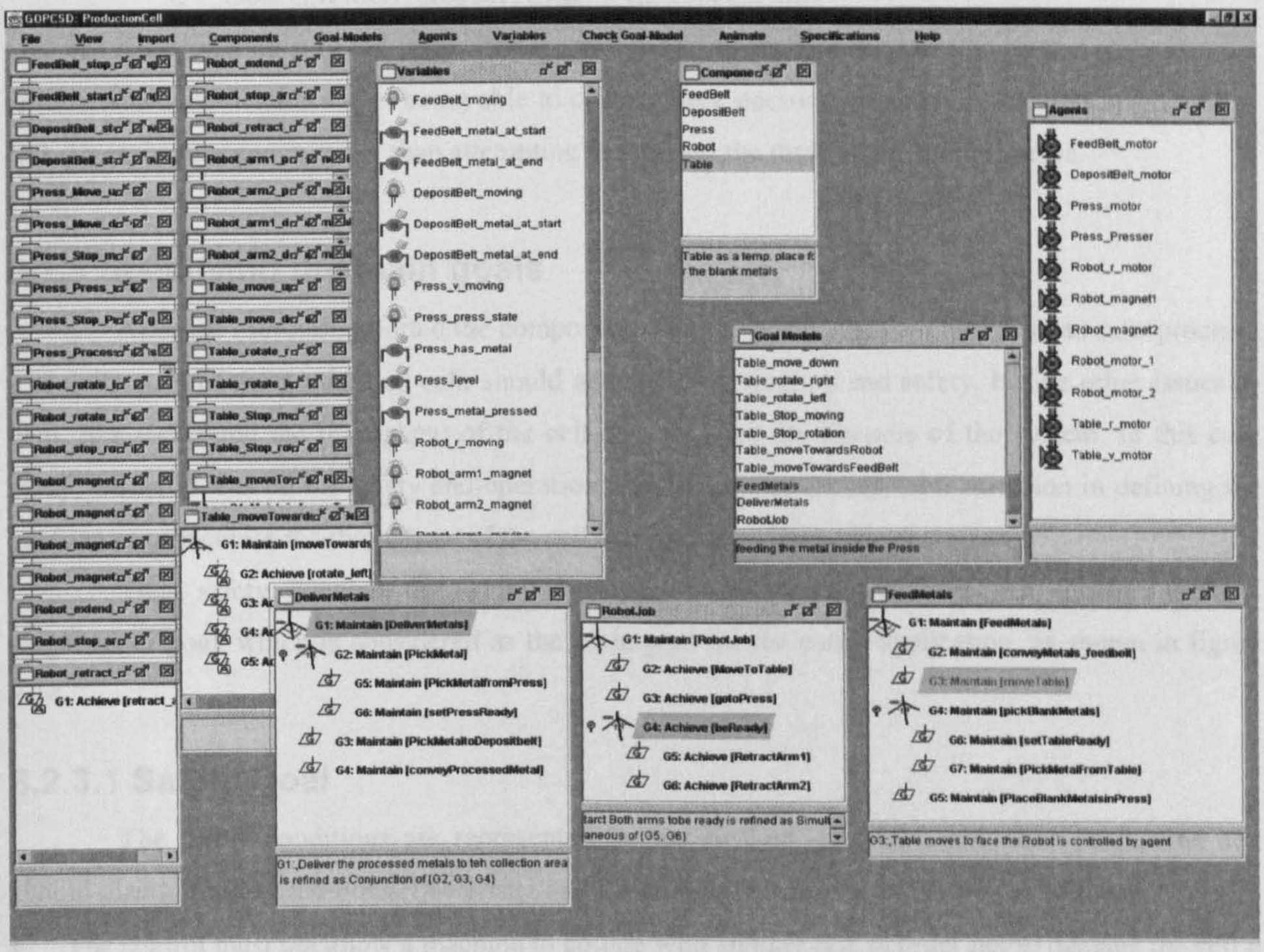


Figure 8.13, the production cell application after importing the templates

8.2.2.3 RobotJob Template

This template involves assigning jobs to the robot; it was not placed in the low-level specifications of the robot component because it is affected by the status of the other components, like the Press and the Table. Since the robot can perform a single service at a time, this high-level goal is refined as a disjunction of the possible services that the robot can accomplish. The robot can collect the processed metals from the press or the unprocessed metals from the rotary table, or be ready to do one of these two services. The *Robot_Job* goal can be refined as a disjunction of the following three goals:

- Goal GM38G2, Move to the Table, if it has a blank metal, facing the robot, the press is empty, and the robot is idle. This goal may also be augmented by generalising the pre-condition to include the case when the press is empty and the table is not ready yet, i.e. to wait for the table.
- Goal GM38G3, Move to the Press, if the processed metal is ready to be collected and the robot is idle.

- Goal GM38G4, BeReady, this goal can be used to enhance the throughput by preparing the robot to be ready to rotate. Since the two arms of the robot can extended or retract separately, this goal can be refined as a simultaneous combination of two goals as follows:
    - Goal GM38G6, Retract Arm1, if the robot is idle.
    - Goal GM38G7, Retract Arm2, if the robot is idle.

One can notice that most of the factors for increasing the throughput are affected by this early abstract requirement specification stage; being able to correct these decisions at the requirements stage will be less expensive and much easier than attempting to increase the throughput at later stages.

8.2.3 Specifying the main goals

After the user has imported the components and templates, the development can now proceed. The main goal of the production cell, should address the operation and safety, beside other issues as well, like increasing the throughput of the cell and ensuring the liveness of the system. In this case study, we will focus on the safety and operation aspects, paying considerable attention in defining the goals so as to increase the throughput of the cell and maintain the liveness.

Thus, safety, liveness, throughput and operation aspects can be combined together under a single super goal, which is considered as the main goal for the entire application, as shown in figure 8.14.

8.2.3.1 Safety Goal

The safety conditions are represented by the abstract goal GM39G2. The production cell should maintain safe conditions [Lewerentz and Lindner 95] during the operation, as follows:

- The control must not allow a machine to collide with another one in order not to damage the cell it self
- The control program must guarantee that two consecutive blanks are always sufficiently separated by a minimum distance to avoid having the two blanks on the table or the press, simultaneously.

This consideration can be represented by an abstract safety goal that can be refined later.

8.2.3.2 Ensure liveness property

The main task requires from the production cell is to press the blank metals and convey them to the collection area; to fulfil this task, we defined the operation goal; however, to ensure that the task will be done properly we define this liveness goal and refine it into sub-goals, which ensure that no metals will be missed or unstamped. Hence, we created the abstract goal GM39G3 to define the liveness assurance. The controller should guarantee the liveness of the system; each metal entered to the system must be processed and delivered to the deposit belt, eventually.

8.2.3.3 Increase the Throughput

The throughput of production-line systems, which process input products or manufacture them, have the throughput as an important attribute that can increase using one particular system over other. Although, the safety and operation goal affects the throughput; it can deter or aid it. However, it

should be helpful to define the throughput goals separately to achieve them. The throughput goal will be represented by the abstract goal GM39G4.

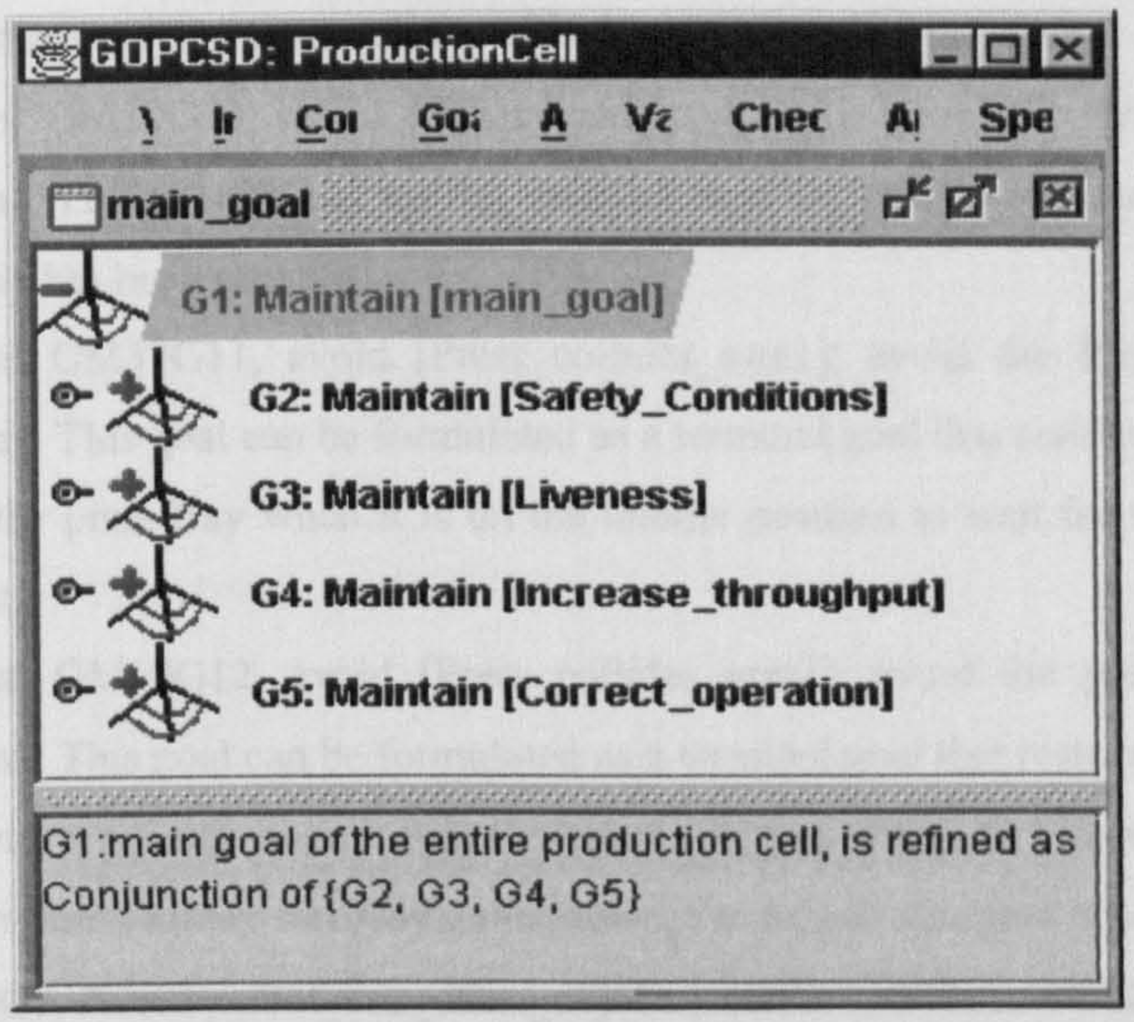


Figure 8.14, the main goal of the production cell

8.2.3.4 Operation goal

The production cell's highest operation goal (the abstract goal GM40G5) should ensure the correct operation of the different components as well as the inter-component interactions.

8.2.3.5 Other Issues

In addition to the safety, liveness, throughput, and operation considerations, other issues can be mentioned in the production cell requirements as in [Lewerentz and Lindner 95], where the design was required to be easy to be adapted to similar production cells, efficient to achieve a high throughput of the blanks through the system, and the cost-benefit ratio should be balanced.

The four aspects (safety, liveness, throughput, and operation) should together define the entire production cell. In case of any conflict between goals from different aspects, the order in which these aspects appear should be considered in preferring one goal to another according to its type.

Some of these issues may be embedded within the operational goals and the safety goals like maximizing the throughput; others can be regarded as non-behaviour requirements [Davis 93] and be judged after looking at the entire application, as we will discuss later.

8.2.4 Goal-model Refinement

After importing the templates and the components and creating the main goals, the tool should guide the user in refining the various goal-models to specify the safety and operational requirements.

8.2.4.1 Refining Safety Goals

The safety goal can be refined as four conjunctive goals, as follows:

- Goal GM39G6, Avoid [collide_the_machines]: avoid collision between the different machines; this goal ensures that the robot arm will not be hit by the table or the press; it

restricts the movement of the press or the table until the robot's appropriate arm retracts first. Thus, this goal can be refined as the two following simultaneous goals to restrict the motor of the press or the rotational motor of the table:

- Goal GM39G10, avoid [Table_collides_Robot]: avoid the table hitting the robot arm1. This goal should restrict the rotation of the table from starting unless the robot arm1 has been retracted.
- Goal GM39G11, avoid [Press_collides_arm1]: avoid the Press hitting the robot arm1. This goal can be formulated as a terminal goal that restricts the vertical motion of the press tray when it is on the middle position to wait for robot arm1 to retract first.
- Goal GM39G12, avoid [Press_collides_arm2]: avoid the press hitting the robot arm2. This goal can be formulated as a terminal goal that restricts the vertical motion of the press tray when it is in the lower position to wait for robot arm2 to retract first.

In case that the table return journey starts by going down, the second sub-goal is not required.

- Goal GM39G7, Avoid [robot_hits_other_objects]: avoid that the robot hits others objects; this might happen if there are existing objects or humans within the cylinder formed by the robot arm as the radius and the difference in vertical level between the two arms; to remove such cases, the robot should not rotate unless it has both arms fully retracted. Thus this goal can be directly formulated as a terminal goal that restricts the rotational motion of the robot.
- Goal GM39G8, avoid [two_metals_on_table]: avoid having two metals on the table at the same time; this goal ensures the table will not have more than one metal by switching the feed belt motor off when the table has a metal.
- Goal GM39G9, avoid [two_metals_in_press]: avoid having two metals inside the press at the same time; this goal can be achieved by forcing the robot arm2 not to drop a metal within the press if arm2 is already extended and has a metal, when the press has metal.

8.2.4.2 Refining the Liveness Goal

As the liveness property addresses the issue that each blank metal entered into the system must be preserved and processed, the main goal can be defined as a conjunction of avoiding goals, each of them prohibiting the loss of the metal at the different stages and the assurance of processing. Thus, the main liveness goal can be refined as the conjunction of the following four goals:

- Goal G39G13, Avoid [dropping_metals]: do not drop metals from robot arms unless over the press or the deposit belt; this goal should assure that the magnet's actuator will keep the metal attached to the appropriate robot arm unless it is convenient to leave it, over the press tray (in the middle position) or over the start of the deposit belt.
- Goal G39G14, Avoid [robot_pushes_metals_out]: do not let the robot pushes the metal away; this goal can be regarded as an assumption from the environment, as we previously assume that when the robot arms are fully extended, the arms' magnet will be just over the press's lower and middle positions or the table's upper position. In this case, the goal can be considered as an environmental goal and will not be translated into the specification.

- Goal G39G15, Avoid [pick_unprocessed_metals]: avoid the robot picking up unprocessed metals; this goal ensures that the robot will not pick unprocessed metal, but rather it will stop functioning; thus this situation can be considered as a detected fault within the press component.
- Goal G39G16, Avoid [metal_drop]: avoid metals dropping from the feed belt; this goal avoids dropping blank metals when the table is not ready to receive them because it is not aligned to the feed belt or not at the same level. The goal will switch the feed belt motor off in either case.

8.2.4.3 Refining the Throughput Goal

As explained earlier, the throughput issue is affected by the four following sub-goals, which are defined as follows:

- Goal GM39G17: Maintain [Feedbelt_on]: keep the feed belt motor on; this goal ensures the stream of metals will be supplied to increase the throughput of the production cell; the goals can be specified as a simple terminal goal that maintains the feed belt motor on.
- Goal GM39G18: Maintain [best_use_the_robot]: make the best use of the robot, keeping it busy or at least ready to serve the next expected job. This goal is one of the imported high-level templates, Robot_Job. We will refine it later.
- Goal GM39G19: Maintain [Press_ready]: keep the press ready after it pressed a metal to press new ones; this goal returns the Press to its middle position as soon as possible after the robot picks up the processed metal. Even though this goal shows some conflict behaviour with the safety goal, we may leave it for the conflict analysis stage to correct it; or, alternatively we should place more restriction on the movement of the table to wait until the robot arm is retracted. This goal can be copied from the imported press's low-level goal-model.
- Goal GM39G20, Maintain [Table_ready]: the table should be ready as soon the blank metal has been picked up; after the robot picks up the blank metal, this goal should return the table to its normal position, where it faces the feeding belt and can accept a new blank metal, as soon as possible. Similar to the press be ready goal, this goal seems to conflict with a safety goal; however, we can leave it for the conflict analysis stage to correct it; or, alternatively, we should place more restriction on the movement of the table to wait until the robot arm retracts. This goal can be copied from the imported table's low-level goal-model.
- Goal GM39G21, Maintain [Depositbelt_on]: keep the feed belt motor on; this goal addresses the deposit belt, which is not controlled by the application. This goal should be considered as non-functional goal.

The refinement for the safety, liveness and throughput aspects are shown in figure 8.16 as goals G2, G3 and G4, respectively, within the goal-model main-goal.

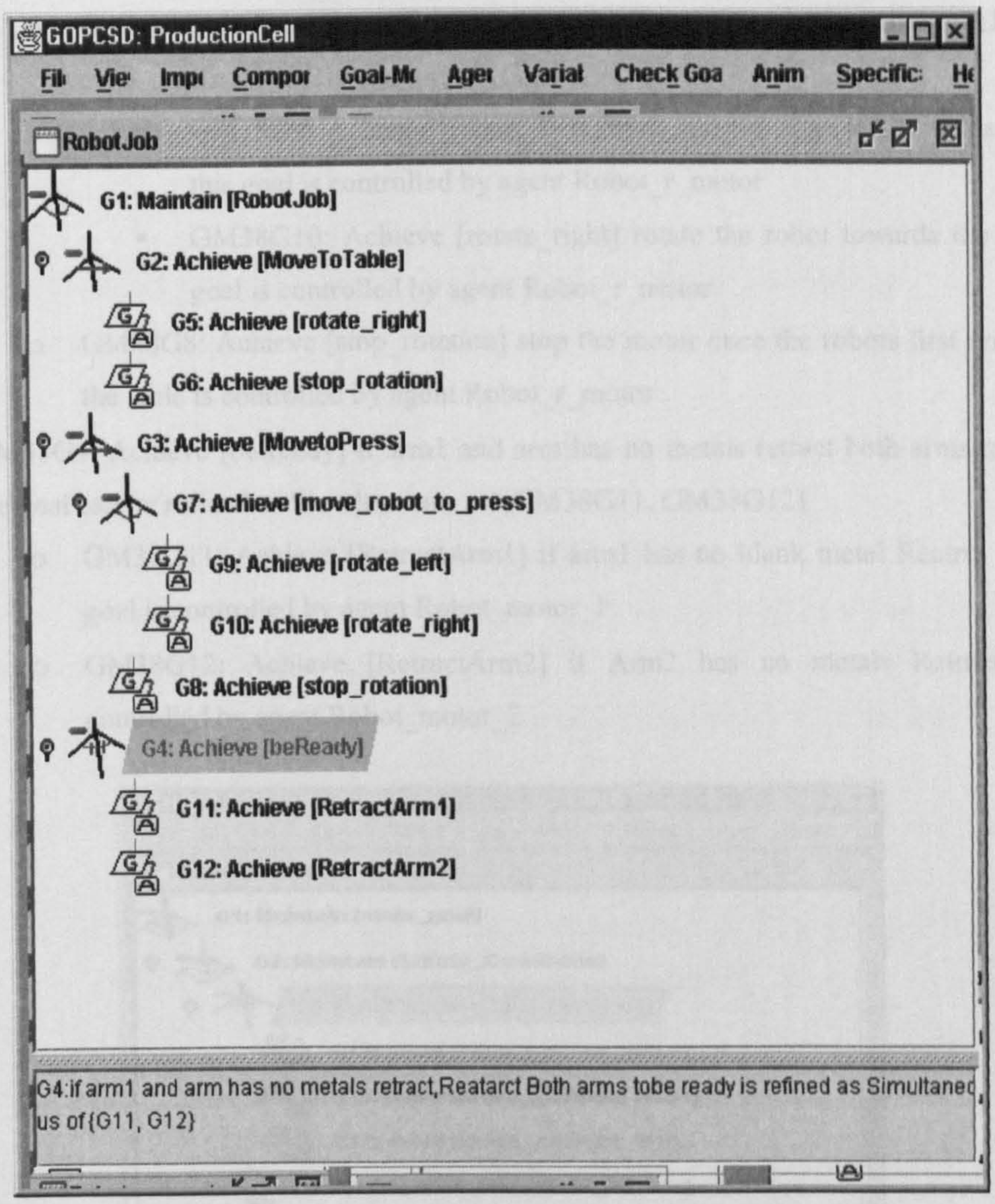


Figure 8.15, the Robot_job template after refinement

8.2.4.2 Refining the Robot_Job Template

The robot job template can be defined as shown in figure 8.15 with the maintain goal GM38G1: Maintain [RobotJob] utilise the robot; this goal can be refined as the disjunction of the three following goals {GM38G2, GM38G3, GM38G4} as follows:

- GM38G2: Achieve [MoveToTable] if Robot is ready and there is blank to be collected from the table moves towards rotary table until arm1 face the table is refined as Sequence of {GM38G5, GM38G6}
  - GM38G5: Achieve [rotate_right]: this goal is a copy of rotate right low-level goal of the robot; it is controlled by the Robot_r_motor agent. This goal rotates the robot until the first arm faces the rotary table.
  - GM38G6: Achieve [stop_rotation] stop the motor once the robots first arm is facing the table is controlled by agent Robot_r_motor
- GM38G3: Achieve [MovetoPress]: if Robot is ready to rotate and there is a processed metal to be collected from the press; the robot rotates towards the Press until arm2 face the Press is refined as Sequence of {GM38G7, GM38G8}

- GM38G7: Achieve [move_robot_to_press] move the robot so that arm1 faces the press is refined as Disjunction of {GM38G9, GM38G10}
  - GM38G9: Achieve [rotate_left] rotate the robot to left towards the press; this goal is controlled by agent Robot_r_motor
  - GM38G10: Achieve [rotate_right] rotate the robot towards the press; this goal is controlled by agent Robot_r_motor
- GM38G8: Achieve [stop_rotation] stop the motor once the robots first arm is facing the table is controlled by agent Robot_r_motor

- GM38G4: Achieve [beReady] if arm1 and arm has no metals retract both arms to be ready; the goal can be refined as Simultaneous of {GM38G11, GM38G12}
  - GM38G11: Achieve [RetractArm1] if arm1 has no blank metal Reatrct Arm1; this goal is controlled by agent Robot_motor_1
  - GM38G12: Achieve [RetractArm2] if Arm2 has no metals Retract Arm2 is controlled by agent Robot_motor_2

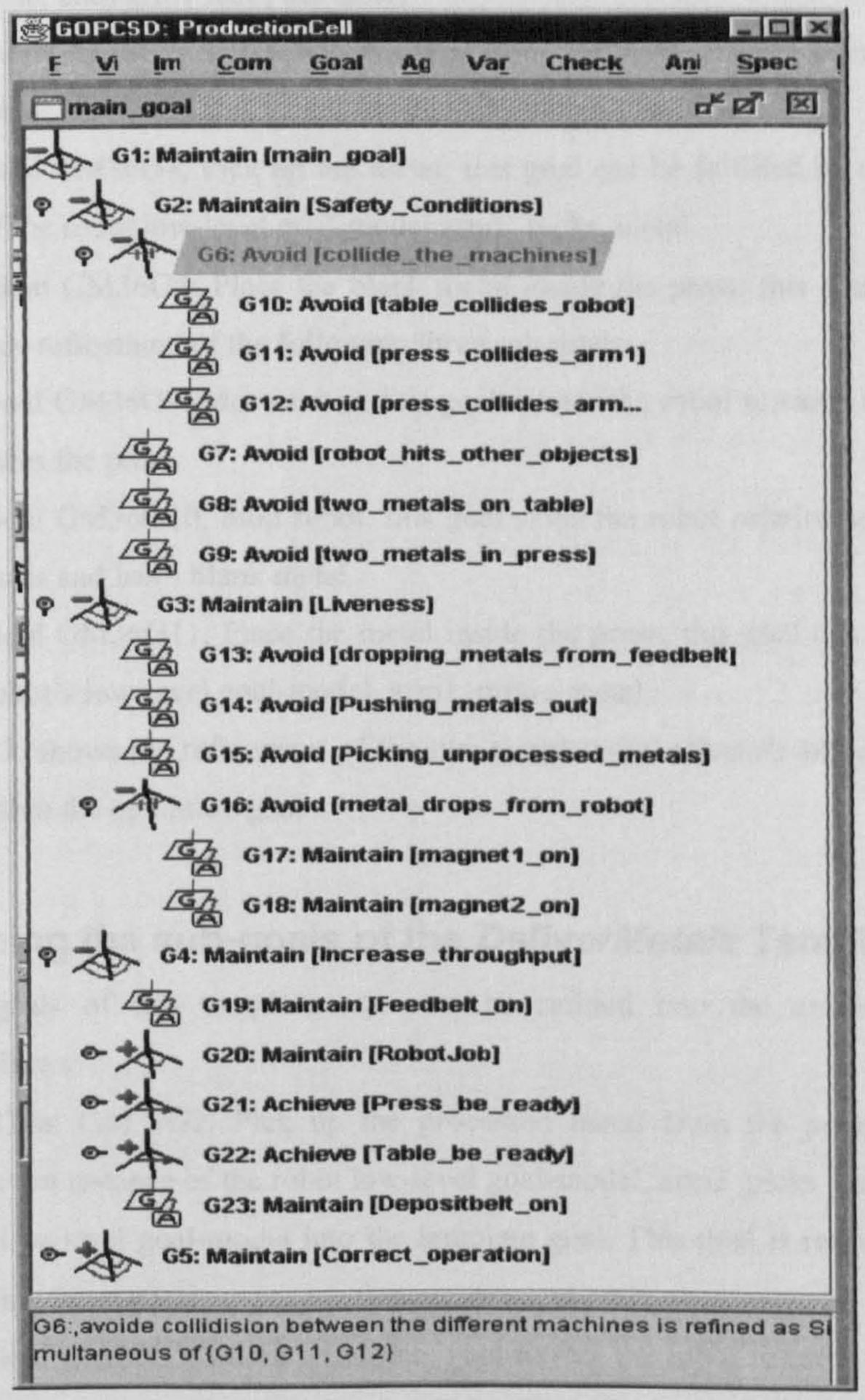


Figure 8.16, the main goal of the production cell after refining safety, liveness and throughput goals

8.2.4.3 Refining the Operational Goal

The operational goal can be refined into the three existing goals *feedMetals*, *processMetals*, and *deliverMetals*. Since the *processMetals* goal has been already refined, within the press component, we refine the other two goals as follows:

8.2.4.3.1 Refining the sub-goals of the *feedMetals* Template

The sub-goals of this template can now be refined into the existing low-level goal-models/goals as follows:

- Refining Goal GM36G2: Convey the metals to the rotary table; this goal can be specified as a terminal goal that maintains the feed belt motor switched on.
- Refining Goal GM36G3: Move the table towards the robot
- Refining Goal GM36G4: Pick up the blank metal; this goal can be refined as the sequence refinement of the following three sub-goals:
    - Goal GM36G6, Move robot; this goal rotates the robot to the Right towards the table, until arm1 faces the table.
    - Goal GM36G7, Stop robot; this goal stops the motor if arm1 is empty and facing the table.
    - Goal GM36G8, Pick up the metal; this goal can be fulfilled by copying an instance of the robot low-level goal-model arm1_picks_metal.
- Refining Goal GM36G5: Place the blank metal inside the press; this goal can be defined as the sequence refinement of the following three sub-goals:
    - Goal GM36G9, Move robot; this goal rotates the robot towards the press until arm1 faces the press.
    - Goal GM36G10, Stop robot; this goal stops the robot rotation when arm1 faces the press and has a blank metal.
    - Goal GM36G11, Place the metal inside the press; this goal can be copied from the robot's low-level goal-model, arm1_drops_metal.

Figure 8.17, shows the refinement of the two templates, *feedmetals* and *delivermetals* before combining them within the operation goal.

8.2.4.3.2 Refining the sub-goals of the *DeliverMetals* Template

The sub-goals of this template can now be refined into the existing low-level goal-models/goals, as follows:

- Refining Goal GM37G2, Pick up the processed metal from the press; this goal can be regarded as an instance of the robot low-level goal-model, arm2_picks_metals. Hence, we can copy this low-level goal-model into the template goal. This goal is refined as a sequence of the following three goals:
    - Goal GM37G5, Move robot; this goal moves the robot towards the press until arm2 faces the press. It is refined as a disjunction of two terminal goals to the rotate to left or right depending on its current angle.

- Goal GM37G6, Stop robot; this goal stops the rotation of the robot when arm 2 faces the press while holding a metal.
- Goal GM37G7, Place the processed metal on the deposit belt; this goal can be copied from arm2_drops_metals.

- Refining Goal GM39G3, Place processed metals at the start of the deposit belt; this goal is refined as a sequence of the following three sub-goals:
  - Goal GM37G8, Move robot; this goal rotates the robot until arm2 faces the start of the deposit belt; this goal can be refined as a disjunction of two terminal goal to rotate the robot to the left or right.
  - Goal GM37G9, Stop robot; this goal is to stop the robot rotation after it has faced the start of the deposit belt.
  - Goal GM37G10, the robot drops the metal; this goal performs the same function as the robot's low-level goal-model, arm2_drops_metals, which means it can be directly copied from the robot low-level details.
- Refining Goal GM39G4, Convey the processed metal to the collection area. This goal can be assumed as a fact about the environment since it relies on the deposit belt motor, which is not controlled by the developed controller.

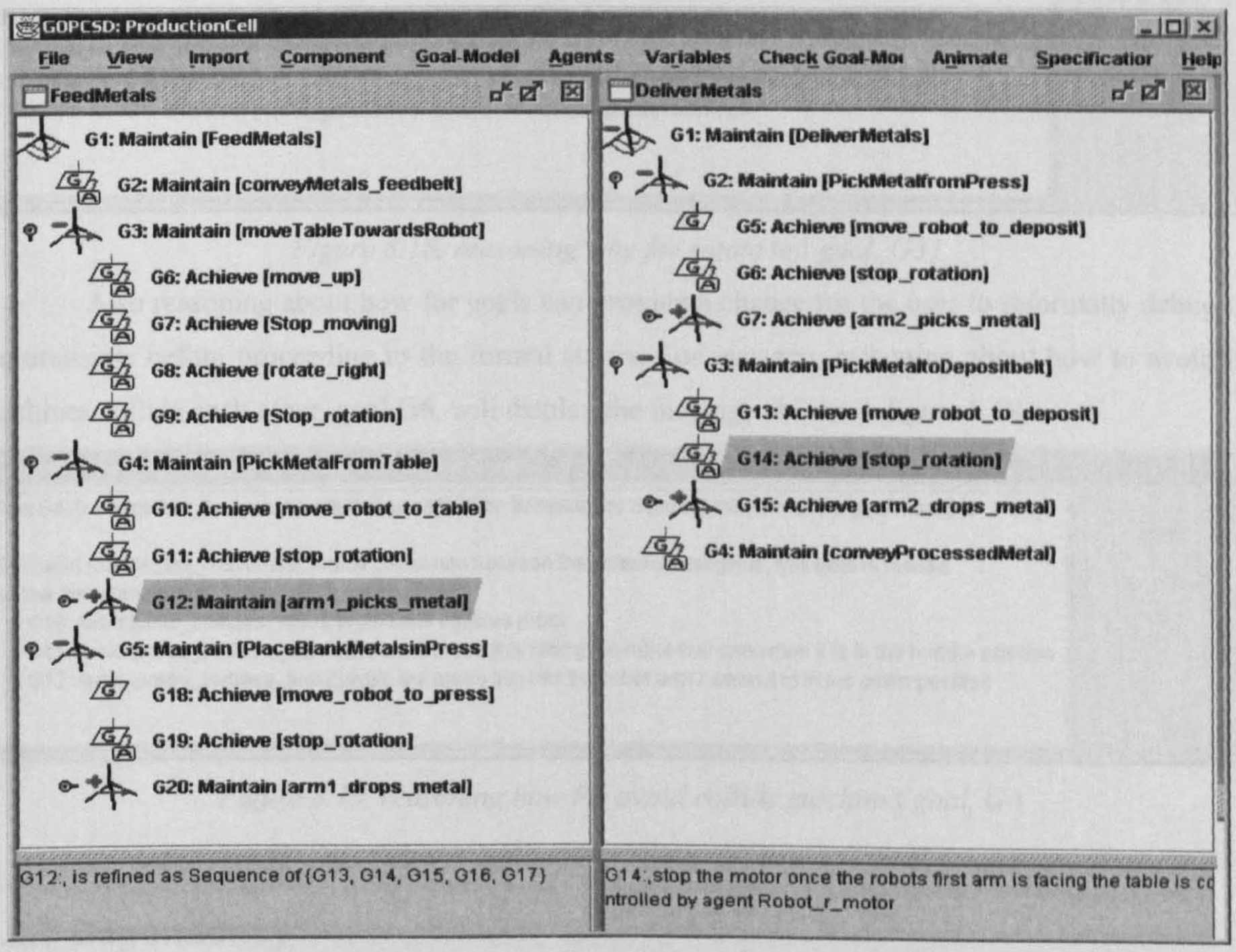


Figure 8.17, the refinement of feedmetals and delivermetals templates

8.2.5 Combining the goal-models

So far, only the operation goal is not fully refined because other template goal-models needs to be combined to represent it; these templates are the *feedMetals*, *processMetals* and *deliverMetals*; although the refinement relationship between these goal-models may appear to be sequence, a deep

look at the sub-goals of these sub-goals should reveal that the sub-goals are addressing different components and can be better (from the throughput aspect) accessed in conjunction.

However, it would be helpful to perform consistency checks on these refined templates before combining them in the main goal-model. Next to this step, we can copy each of these goal-models and then paste them as new child goals of the operation goal. After this step, it may be useful to get rid of the separate goal-models and the other low-level goals if they will not be required later.

Some notes some of the low-level goal-models saved us from using the details of the component, as in the cases when we used the picking or dropping sequence of the robot's arms.

8.2.6 Reasoning about the goals

Before start checking the goal-models formally, it would be helpful to reason how and why about for the important goals; or at least for the goals, which the user does not have full confidence in them. For example, reasoning about why for goal G31 of the *maingoal* goal-model (this goal is one of the increase throughput goals), a message similar to the one shown in figure 8.18 will be displayed.

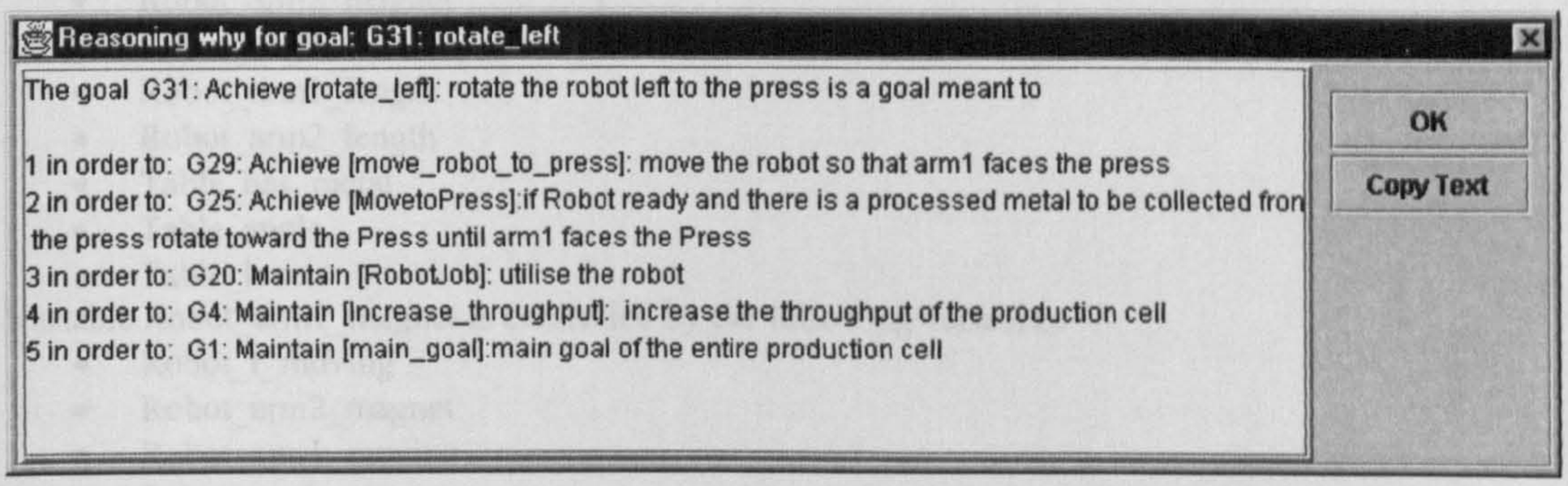


Figure 8.18, reasoning why for rotate left goal, G31

Also reasoning about how for goals can provide a chance for the user to informally debug the requirements before proceeding to the formal stages. For instance, reasoning about how to avoid the machines collide each other, goal G6, will display the message shown in figure 8.19.

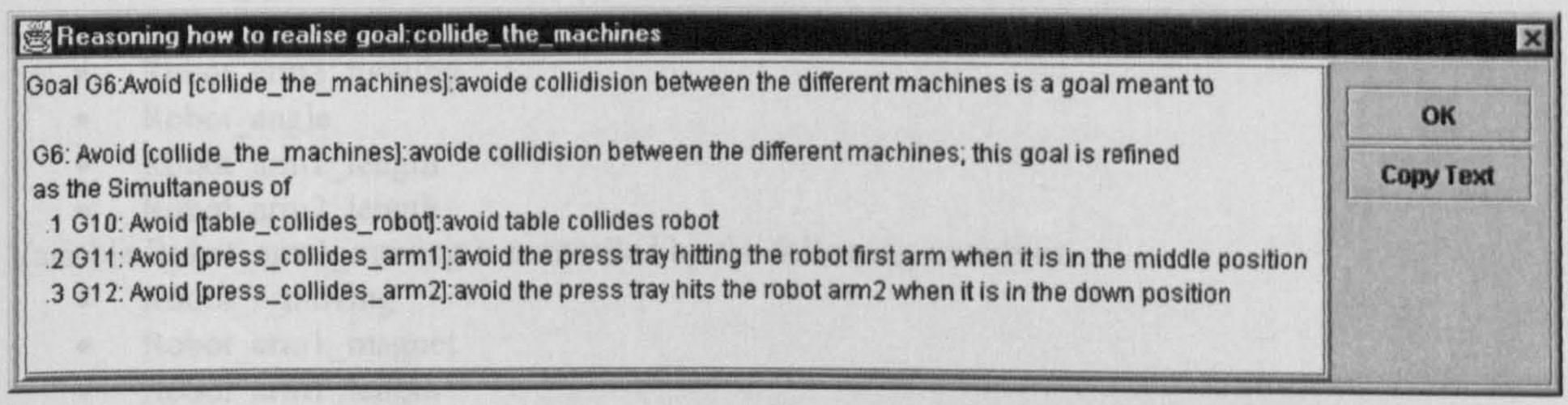


Figure 8.19, reasoning how for avoid collide machines goal, G6

8.2.7 Dependency

An important issue provided by the tool is that it has ability to trace the dependency between the different variables (from controlling the output variables formulae). The tool can establish the effective variables for each output variable, and hence better utilise the space to perform the different checks. Before performing the completeness and consistency check on the main-goal model, we obtained the following "output variable"/variable dependencies:

Variable FeedBelt_moving is controlled by the following variables:
- Table_has_metal
- Table_angle
- Table_level

Variable DepositBelt_moving is not controlled!
Variable Press_v_moving is controlled by the following variables:
- Press_press_state
- Press_has_metal
- Press_level
- Press_metal_pressed

Variable Press_press_state is controlled by the following variables:
- Press_v_moving
- Press_has_metal
- Press_level
- Press_metal_pressed

Variable Robot_r_moving is controlled by the following variables:
- Press_has_metal
- Press_level
- Robot_arm1_magnet
- Robot_arm2_magnet
- Robot_angle
- Robot_arm1_length
- Robot_arm2_length
- Table_has_metal
- Table_angle
- Table_level

Variable Robot_arm1_magnet is controlled by the following variables:
- Robot_r_moving
- Robot_arm2_magnet
- Robot_arm1_moving
- Robot_angle
- Robot_arm1_length

Variable Robot_arm2_magnet is controlled by the following variables:
- Press_has_metal
- Press_level
- Robot_r_moving
- Robot_arm1_magnet
- Robot_arm2_moving
- Robot_angle
- Robot_arm1_length
- Robot_arm2_length

Variable Robot_arm1_moving is controlled by the following variables:
- Robot_r_moving
- Robot_arm1_magnet
- Robot_arm1_length

Variable Robot_arm2_moving is controlled by the following variables:
- Robot_r_moving
- Robot_arm2_magnet
- Robot_arm2_length

Variable Table_v_moving is controlled by the following variables:
- Table_has_metal
- Table_angle
- Table_level
- Table_r_moving

Variable Table_r_moving is controlled by the following variables:
- Robot_arm1_length
- Table_has_metal

- Table_angle
- Table_level
- Table_v_moving

Comments on the dependency

- We can identify that there is no dependency between the variables of the nonadjacent components, such as the Table and Press or the Feed Belt and the Robot. On the other hand, the Robot rotation is affected by the table and press variables. The robot arms are affected by the table and the press states.
- It should be noticed that, the deposit belt motor is not controlled by the developed application. This enables the user to remove the deposit belt agents and variables from the application agent and variable list, respectively.
- This analysis can provide considerable guidance in building the different controllers for the various parts of the production cell application, especially in structuring the generated specifications of the B machines.
- In fact, this analysis may be used to reveal how much the crucial design aspects like modality, clarity and coupling are fulfilled.

8.3 Checking and validating the requirements

After constructing the goal-model of the production cell, the tool should guide the user to capture the requirements bugs that may still exist within the specified requirements.

8.3.1 Checking the correctness of the goal-model structure

Before checking the formal parts of the goal-model, user can debug the goal-model by checking the correctness of the goal-model and whether each functional terminal goal has been assigned to an agent and whether the refinement constraints (that we have mentioned in chapter 6) are met. Having checked the *feedmetals*, *delivermetals* and *RobotJob* template goal-models and the *maingoal* goal-model, we found that none of them violates the structural constraints.

8.3.2 Obstacle Analysis

It should be useful, especially in medium- or large-scale systems, to perform the obstacle analysis as early as possible so the goal-model can be modified and prepared for the next checks, such as performing completeness and conflict checks. Obstacle analysis requires further attention from the user, in considering other conditions that can stop the goal from being accomplished.

For example, in goal-model *maingoal*, goal G19, which maintains the feed belt motor switched on to increase the throughput, will be obstructed when the table is not ready to receive new blank metals or when it is facing the feed belt and is at the same level as the feed belt but already has a metal. In these two cases, the motor cannot be kept on otherwise metals will drop from the belt or two metals will be placed on the table, which is not a safe condition.

These two conditions can be considered as two obstacles that occur when the pre-condition of the obstructed goal is satisfied and the negation of its post-condition is negated i.e. *feedbelt_motor = OFF*. These two obstacles cannot be removed.

However, the *metal_at_end* sensor's reading of the feed belt component can be used to attenuate the effect of these two obstacles; i.e. to know whether there is a coming metal on the feed belt so the control program needs to switch the feed bet motor off or not.

In order to document this obstacle for later analysis or just for documentation, the GOPCSD tool enables the user first to define the obstacles as shown from figures 8.20 and 8.21.

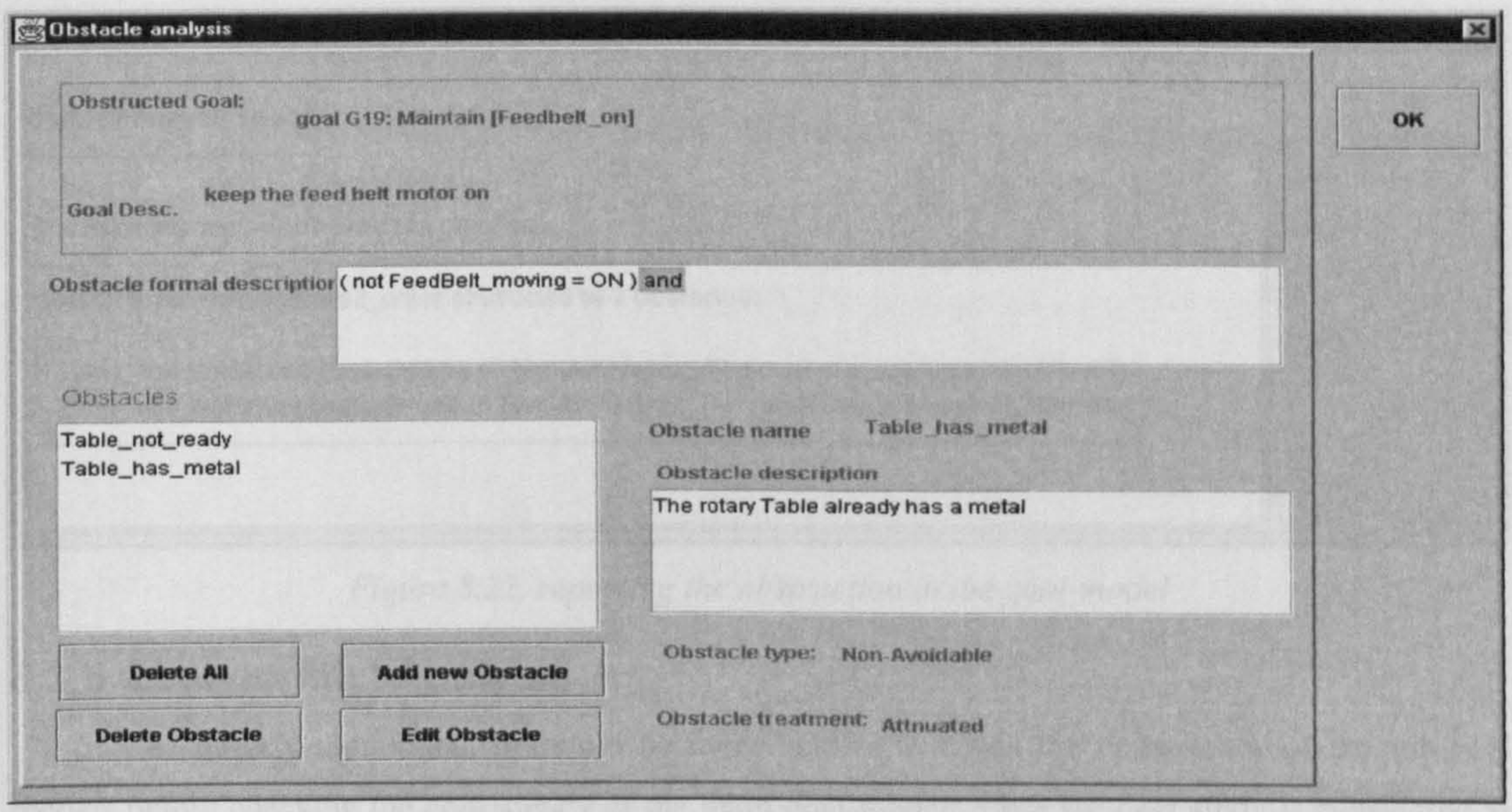


Figure 8.20, introducing obstacles for goal G19 (increase the throughput)

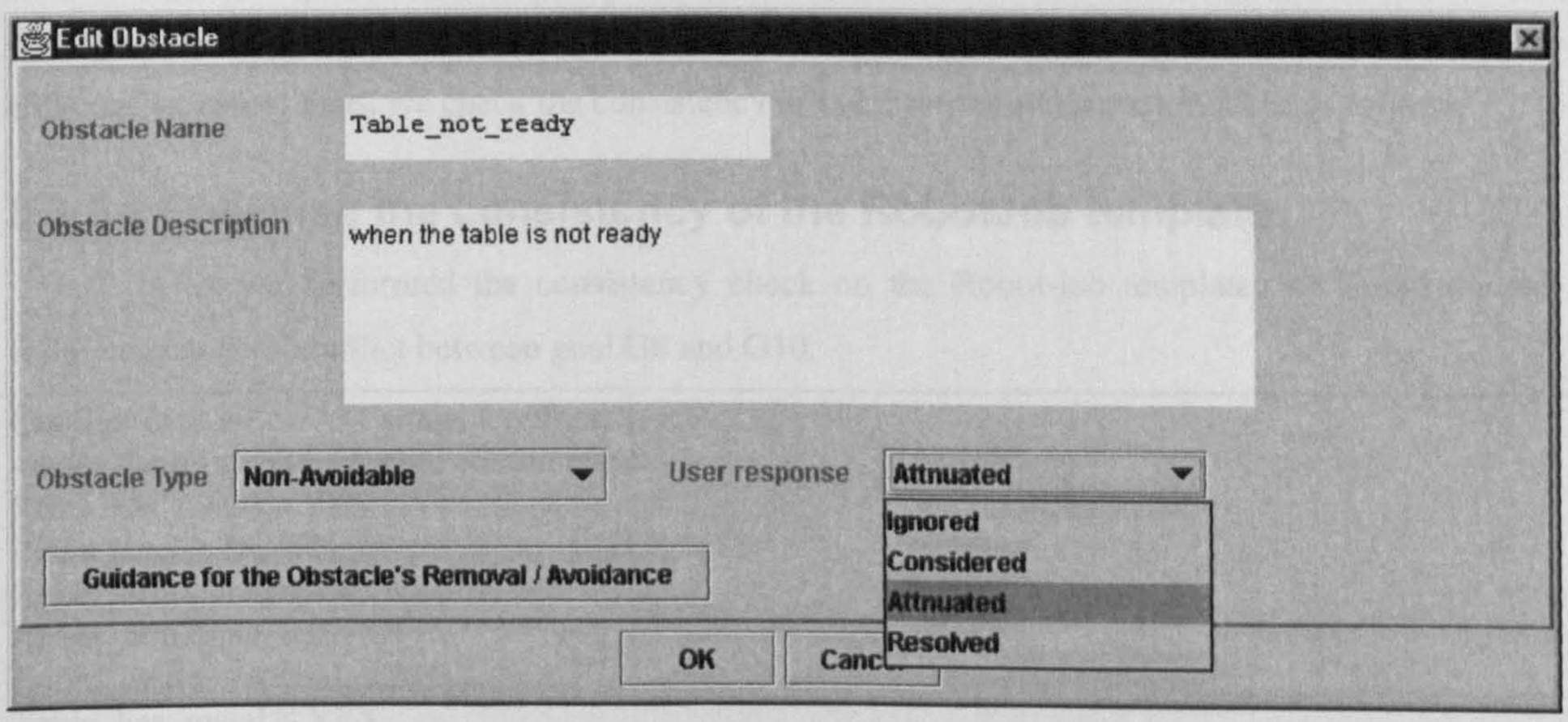


Figure 8.21, updating the status of the obstacle after modifying the obstructed goal

After the user defines the obstacle, he/she has to decide the type of the obstacle non-avoidable (like this case) or removable. And hence, the goal may be modified or a new goal can be added to resolve the obstacles. Then the user should modify the current state of each obstacle as shown in figure 8.21, where the user changes the user-response to *attenuated* after modifying goal G19.

The pre-condition of goal G19 may be strengthened to (there is no metal coming) or (the table does not have a metal and it is in the position to receive new metals), so the formal description of the goal will be:

(Feedbelt_metal_at_end = NO) or (Table_has_metal = NO and Table_level = DOWN and *Table_angle = FEED_BELT_FACING) => Feedbelt_motor = ON*

Finally, the user should generate a final obstruction report for documenting the obstacles of the main goal-model, as shown in figure 8.22.

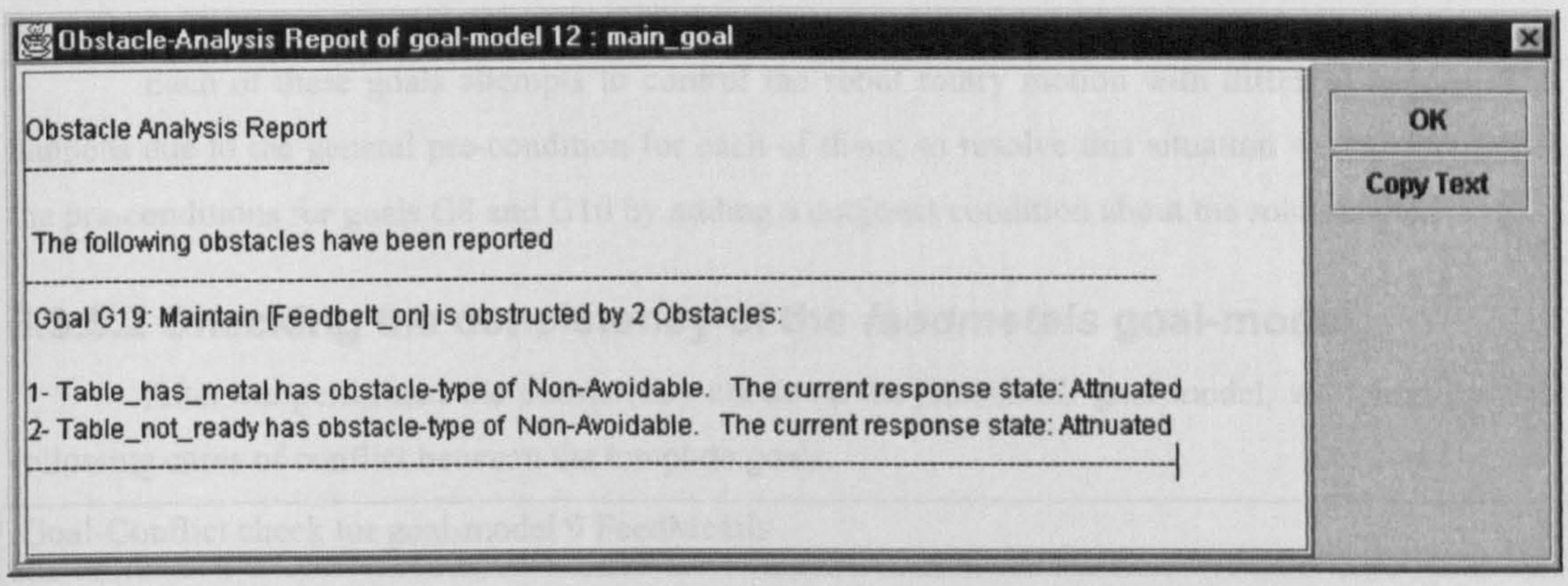


Figure 8.22, reporting the obstruction in the goal-model

8.3.5 Goal-conflict Analysis

As already mentioned, it should be more helpful to check the consistency of the sub-goal models before checking the consistency of the main goal-model. Since the case study is considered as medium scale, we can focus firstly on the critical conflicts, which are considered far more important than the soft conflicts, as mentioned earlier in chapter 6. The GOPCSD tool enables the user to hide the soft conflict cases. Thus, we check the consistency of the three template goal-models, as follows.

8.3.5.1 Checking the consistency of the RobotJob template

After we performed the consistency check on the Robot-job template, we found the two following cases of conflict between goal G8 and G10.

Conflict case no1: Critical Conflict
under the following variable combination:
Press_has_metal = YES
Press_level = DOWN
Robot_arm1_magnet = OFF
Robot_arm2_magnet = OFF
Robot_angle = ARM1_TABLE
Table_has_metal = YES
Table_angle = ROBOT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 2 goals :
1-G10(… =>Robot_r_moving = RIGHT) Agent: Robot_r_motor
2-G8(…=>Robot_r_moving = STOP) Agent: Robot_r_motor
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict

---

Conflict case no2: Critical Conflict
under the following variable combination:
Press_has_metal = YES

```
Press_level = DOWN
Robot_arm1_magnet = ON
Robot_arm2_magnet = OFF
Robot_angle = ARM1_TABLE
Table_has_metal = YES
Table_angle = ROBOT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 2 goals :
1-G10(...=>Robot_r_moving = RIGHT ) Agent: Robot_r_motor
2-G8(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
---------------------------------------------------------------------------
2 Conflict cases have been reported!
```

Each of these goals attempts to control the robot rotary motion with different actions. This happens due to the general pre-condition for each of them; to resolve this situation we can strengthen the pre-conditions for goals G8 and G10 by adding a conjunct condition about the robot angle.

8.3.5.2 Checking the consistency of the *feedmetals* goal-model

After we performed the consistency check on the *feedmetals* goal-model, we found the four following cases of conflict between the template goals.

```
Goal-Conflict check for goal-model 9 FeedMetals
---------------------------------------------------------------------------
 Conflict case no1:       Critical Conflict
 under the following variable combination:
Robot_arm1_magnet = ON
Robot_angle = ARM1_TABLE
Table_has_metal = YES
Table_angle = ROBOT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 2 goals :
1-G22(...=>Robot_r_moving = RIGHT ) Agent: Robot_r_motor
2-G19(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
---------------------------------------------------------------------------
 variable Robot_arm1_magnet needs36 cases
 variable Robot_arm1_moving needs36 cases
 Conflict case no2:       Critical Conflict
 under the following variable combination:
Robot_r_moving = STOP
Robot_arm1_magnet = OFF
Robot_arm1_length = RETRACTED
The variable Robot_arm1_moving is being accessed by 3 goals :
1-G13(... =>Robot_arm1_moving = EXTEND ) Agent: Robot_motor_1
2-G17(...=>Robot_arm1_moving = STOP ) Agent: Robot_motor_1
3-G27(... =>Robot_arm1_moving = STOP ) Agent: Robot_motor_1
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
---------------------------------------------------------------------------
 Conflict case no3:       Critical Conflict
 under the following variable combination:
Robot_r_moving = STOP
Robot_arm1_magnet = ON
Robot_arm1_length = RETRACTED
The variable Robot_arm1_moving is being accessed by 3 goals :
1-G17(...=>Robot_arm1_moving = STOP ) Agent: Robot_motor_1
2-G23(...=>Robot_arm1_moving = EXTEND ) Agent: Robot_motor_1
3-G27(...=>Robot_arm1_moving = STOP ) Agent: Robot_motor_1
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
```

Conflict case no4: Critical Conflict
under the following variable combination:
Robot_r_moving = STOP
Robot_arm1_magnet = ON
Robot_arm1_length = EXTENDED
The variable Robot_arm1_moving is being accessed by 3 goals :
1-G14(...=>Robot_arm1_moving = STOP) Agent: Robot_motor_1
2-G16(...=>Robot_arm1_moving = RETRACT) Agent: Robot_motor_1
3-G24(... =>Robot_arm1_moving = STOP) Agent: Robot_motor_1
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict

---

4 Conflict cases have been reported!

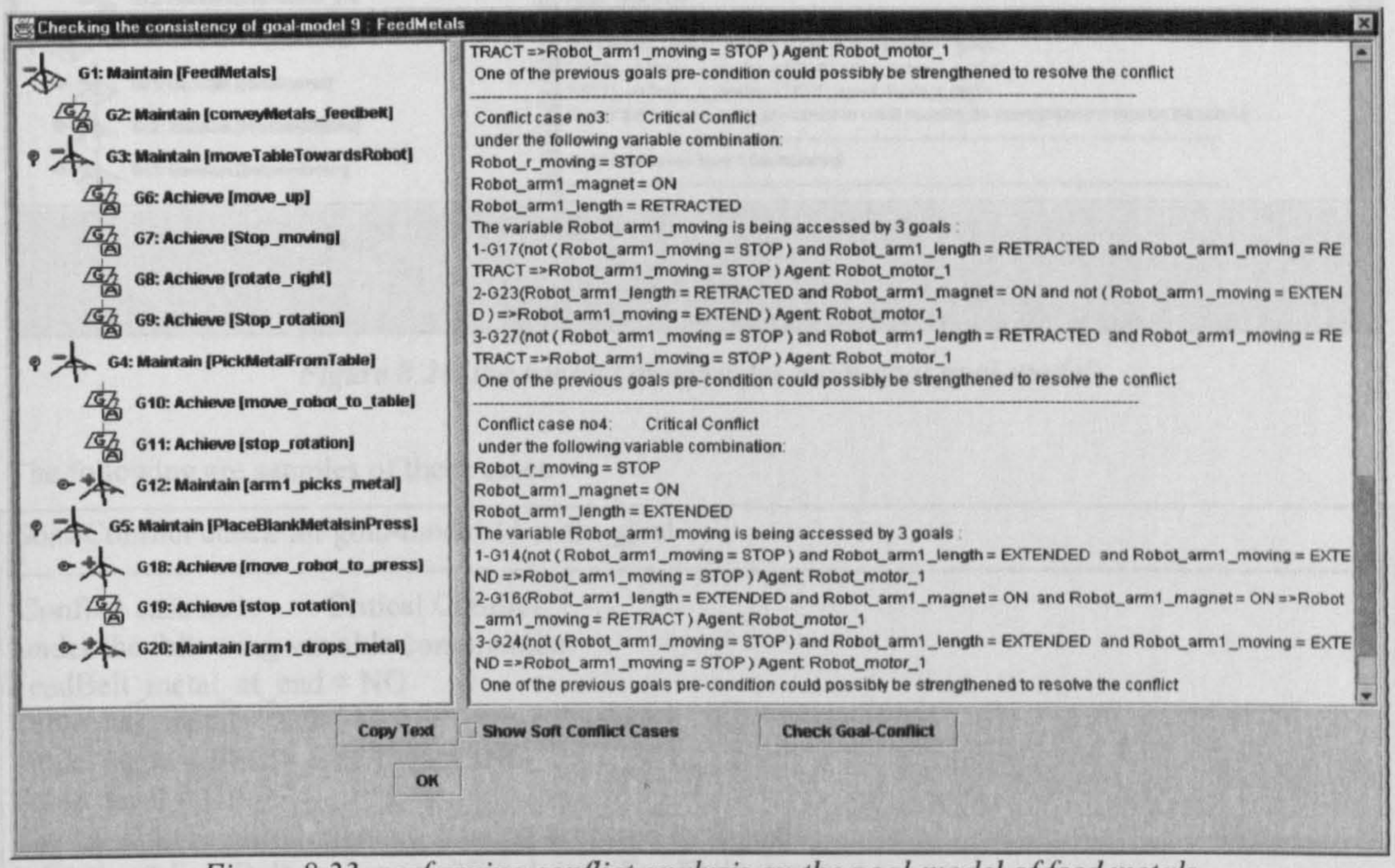


Figure 8.23, performing conflict analysis on the goal-model of feed metals

The pre-conditions of these inconsistent goals need to be strengthened by explicitly using the value of the robot angle (to distinguish between the different situations). So this conflict case, which has three variables, will be split in to a number of consistent cases where the number of effective variables will be four, after adding the *Robot_angle* variable.

8.3.5.3 Checking the consistency of the *delivermetals* template

Like the *feedmetals* goal-model, after performing the conflict analysis for the *delivermetals* goal-model, we found four cases of inconsistency; we also added the *Robot_angle* variable to the pre-conditions of these goals so to remove the conflicts.

8.3.5.4 Checking the consistency of the *maingoal* goal-model

Having modified the sub goal-models of the *Robot_Job*, *feedmetals* and *delivermetals* goal-models. The goal-model of the main goal can now be regarded as complete. And hence, an overall consistency check can be applied. As shown in figure 8.24, the consistency analysis dialogue box reported 56 conflict cases. Although they are all of the type critical conflict, examining these cases and

the inconsistent goals and the variable combinations reveal that the safety, liveness and throughput goals: G7, G9, G10, and G19: of the main-goal goal-model always appear in each cases.

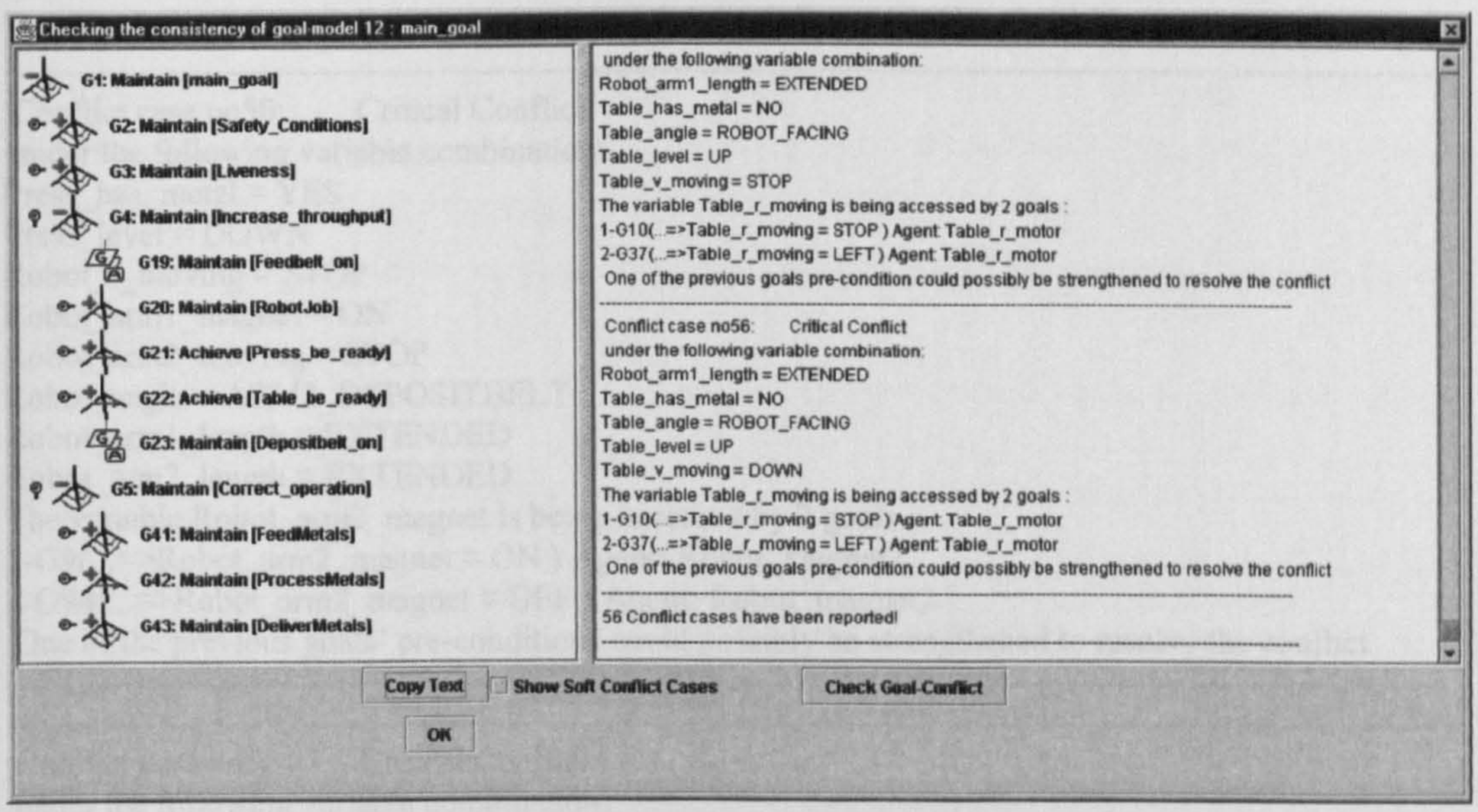


Figure 8.24, the conflict analysis for main-goal goal-model

The following are samples of these cases

```
Goal-Conflict check for goal-model 11 main_goal
----------
 Conflict case no1:       Critical Conflict
 under the following variable combination:
FeedBelt_metal_at_end = NO
Table_has_metal = NO
Table_angle = FEED_BELT_FACING
Table_level = UP
The variable FeedBelt_moving is being accessed by 2 goals :
1-G13(...=>FeedBelt_moving = OFF) Agent: FeedBelt_motor
2-G19(...=>FeedBelt_moving = ON ) Agent: FeedBelt_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
----------
Too many conflict cases for variable: FeedBelt_moving check has been terminated!
----------
Conflict case no35:       Critical Conflict
 under the following variable combination:
Press_has_metal = YES
Press_level = DOWN
Robot_arm1_magnet = ON
Robot_arm2_magnet = OFF
Robot_angle = ARM1_TABLE
Robot_arm1_length = RETRACTED
Robot_arm2_length = RETRACTED
Table_has_metal = NO
Table_angle = FEED_BELT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 4 goals :
1-G7(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
2-G31(...=>Robot_r_moving = LEFT ) Agent: Robot_r_motor
3-G63(...=>Robot_r_moving = LEFT ) Agent: Robot_r_motor
4-G83(...=>Robot_r_moving = LEFT ) Agent: Robot_r_motor
```

```
One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
----------
Too many conflict cases for variable: Robot_r_moving check has been terminated!
----------
 Conflict case no50:       Critical Conflict
 under the following variable combination:
Press_has_metal = YES
Press_level = DOWN
Robot_r_moving = STOP
Robot_arm1_magnet = ON
Robot_arm2_moving = STOP
Robot_angle = ARM2_DEPOSITBELT
Robot_arm1_length = EXTENDED
Robot_arm2_length = EXTENDED
The variable Robot_arm2_magnet is being accessed by 2 goals :
1-G9(...=>Robot_arm2_magnet = ON ) Agent: Robot_magnet2
2-G94(...=>Robot_arm2_magnet = OFF ) Agent: Robot_magnet2
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
----------
 Conflict case no56:       Critical Conflict
 under the following variable combination:
Robot_arm1_length = EXTENDED
Table_has_metal = NO
Table_angle = ROBOT_FACING
Table_level = UP
Table_v_moving = DOWN
The variable Table_r_moving is being accessed by 2 goals :
1-G10(...=>Table_r_moving = STOP ) Agent: Table_r_motor
2-G37(...=>Table_r_moving = LEFT ) Agent: Table_r_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
----------
56 Conflict cases have been reported!
----------
```

The user may have expected to have these conflict cases, because of the conditions that have been imposed on the safety, throughput and liveness goals. These goals now operate as constraints to guide how the operational goals of the production cell should be accomplished, similar to imposing the B invariants on the operations.

8.3.6 Modifying the goal-model and repeating the conflict check

Thus, to remove these inconsistency cases, and because the conflicting goals having different aspects; we should prefer the goals which addressing the safety, liveness, increase the throughput and then the operation, receptively. Therefore, the less important-aspect goal will be strengthened to ensure its condition will be exclusive to the more-important aspect goal.

For example, goals G37 and G10 are conflicting. Goal G10 restricts the rotation of the robot when any of the two arms is extended while goal G37 attempts to rotate the Robot to increase the throughput. Knowing that goal G37 has a throughput aspect and goal G10 has a safety aspect, we should modify the pre-condition of the throughput goal G37 to make sure the two goals will not be active at the same time. The pre-condition of goal G37 can be modified by adding the negation of the

pre-condition of goal G10. Thus, goal G37 will be restricted to be active only when the robot arms are retracted. Similarly, we can modify the other conflicting goals.

After modifying these goals and performing the conflict test again, the following goal pairs were prescribing inconsistent behaviours: (G11, G70), (G12, G35), (G64, G90), (G61, G89), (G63, G30), (G63, G80), (G52, G90) and (G52, G89).

Similarly, we can modify the goals with less important-aspect to satisfy the safety or liveness constraints.

After the second modification of the goal-model the requirements were checked and only two cases have been reported.

```
Goal-Conflict check for goal-model 12 main_goal
---------------------------------------------------------------------------------------------------
Conflict case no1:      Critical Conflict
 under the following variable combination:
Press_has_metal = YES
Press_level = DOWN
Robot_arm1_magnet = OFF
Robot_arm2_magnet = OFF
Robot_angle = ARM1_TABLE
Robot_arm1_length = RETRACTED
Robot_arm2_length = RETRACTED
Table_has_metal = YES
Table_angle = ROBOT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 3 goals :
1-G31(...=>Robot_r_moving = LEFT ) Agent: Robot_r_motor
2-G53(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
3-G83(...=>Robot_r_moving = LEFT ) Agent: Robot_r_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
---------------------------------------------------------------------------------------------------
 Conflict case no2:      Critical Conflict
 under the following variable combination:
Press_has_metal = YES
Press_level = DOWN
Robot_arm1_magnet = OFF
Robot_arm2_magnet = OFF
Robot_angle = ARM2_PRESS
Robot_arm1_length = RETRACTED
Robot_arm2_length = RETRACTED
Table_has_metal = YES
Table_angle = ROBOT_FACING
Table_level = UP
The variable Robot_r_moving is being accessed by 3 goals :
1-G30(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
2-G52(...=>Robot_r_moving = RIGHT ) Agent: Robot_r_motor
3-G80(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 One of the previous goals' pre-conditions could possibly be strengthened to resolve the conflict
---------------------------------------------------------------------------------------------------
2 Conflict cases have been reported!
---------------------------------------------------------------------------------------------------
```

The two cases share the situation where the rotary table has a metal and is waiting for the robot to pick it up and also the press has a processed metal and is waiting for the robot to pick it up.

However, case 1 addresses the situation when the robot is facing the table and case 2 addresses the situation when the robot is facing the press. In case one, it is better to pick up the processed metal, i.e. weakening goal G53.

But, in case 2, it is better to weaken goal G52 so the robot can stop and then extend arm 2 to pick the processed metal from the press. We can achieve this weakening for both goals easily by weakening pre-condition of the common parent goal G46, which concerns with picking up the metal from the table. We restrict goal G46 now to be active only if the press has no metal.

As we can see resolving the conflicts provide the systems engineer with two following advantages:

- Better understanding for the application operation
- Imposing the safety and liveness constraints on the operation goals
- Resolve the conflicts that exist a result of the imperfect design of the requirements

8.3.7 Reachability Check

After we have performed the reachability test on goal-model *main_goal,* the following unreachable goals have been reported.

```
Unreachable Goals Analysis for goal-model 12 main_goal
---------------------------------------------------------------------------------------
the following goals are unreachable:
G15: Picking_unprocessed_metals
G36: Stops
G58: ARM1_RETRACTS
G59: stop_arm1
G67: ARM1_DROPS
G68: ARM1_RETRACTS
G69: stop_arm1
G71: stoptray
G95: ARM2_RETRACTS
G96: stop_arm2
Each of these goals possibly been placed under inappropriate parent-goal,
its pre-condition conflicts with one of its ancestor goals,
or its pre-condition conflicts with one of its predecessor-goals in case of sequence refinement
---------------------------------------------------------------------------------------
```

In each of these cases, there is a logical contradiction between the accumulated pre-condition (chapter 5) of the unreachable goals. The pre-condition may contradict with one of the ancestor goals' pre-conditions or with the predecessor goal's post-condition (in case of sequence patterns). Usually, in large systems there is chance to have such errors, and hence the tool can detect these cases.

8.3.8 Completeness Check

Unlike the consistency issue, the completeness check needs to be performed on the entire goal-model since it involves the output variables' determinacy. Although, in some variable combinations, the user may decide that these situations will not happen, the user still needs to be informed of these situations, so he/she can take critical decisions like shutting the system down.

After checking the completeness of the goal-model *main_goal,* each of the output variables was examined against the effective variables, which are responsible of assigning values to it. When the

output variable value is not fully determined from these variables or it depends on its initial value, the tool reports an incompleteness case.

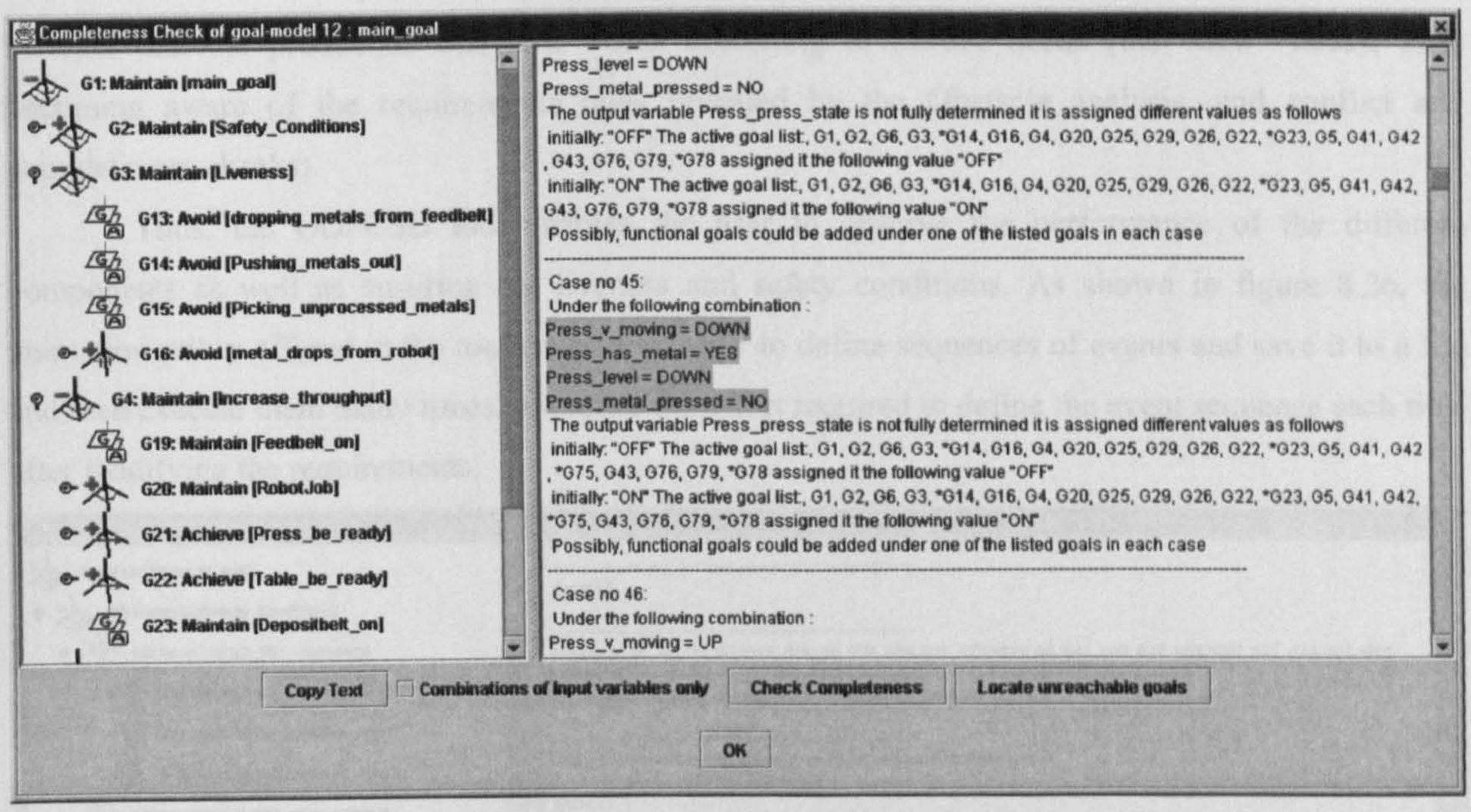


Figure 8.25, checking the completeness of the main_goal

As shown in figure 8.25, the completeness analysis dialogue box shows the reported cases. 235 incompleteness cases were reported; for example, the following are samples of these cases:

Completeness check for goal-model 12 main_goal

Case no 21:

Under the following combination :
Press_press_state = OFF
Press_has_metal = YES
Press_level = MIDDLE
Press_metal_pressed = YES
The output variable Press_v_moving is not fully determined it is assigned different values as follows

Case no 233:

Under the following combination:
Robot_arm1_length = RETRACTED
Robot_arm2_length = RETRACTED
Table_has_metal = YES
Table_angle = FEED_BELT_FACING
Table_level = DOWN
Table_v_moving = STOP
The output variable Table_r_moving is not fully determined

- Case 21 concerns the press motor when the press tray is in the middle position, but the metal is already pressed. Although, this is an impossible situation, it may indicate a fault in the press sensors.
- Case 233 reports that the Table rotation will not be determined when it is down and facing the robot; this is an unreachable position according to our analysis in the table component.

However, the user can include these cases by generalising the actuating goals to consider these impossible situations as well.

8.3.9 Animating the goal-model

After the user performs the conflict and completeness checks, he/she probably needs to validate that the production cell still works according to his/her needs (the "new" needs, after becoming aware of the requirements bugs revealed by the Obstacle analysis, and conflict and completeness checks).

Thus, the GOPCSD tool enables the user to observe the performance of the different components as well as ensuring the liveness and safety conditions. As shown in figure 8.26, the animation utility offered in the tool enables the user to define sequences of events and save it to a file and then execute them many times, to reduce the effort required to define the event sequence each time after modifying the requirements.

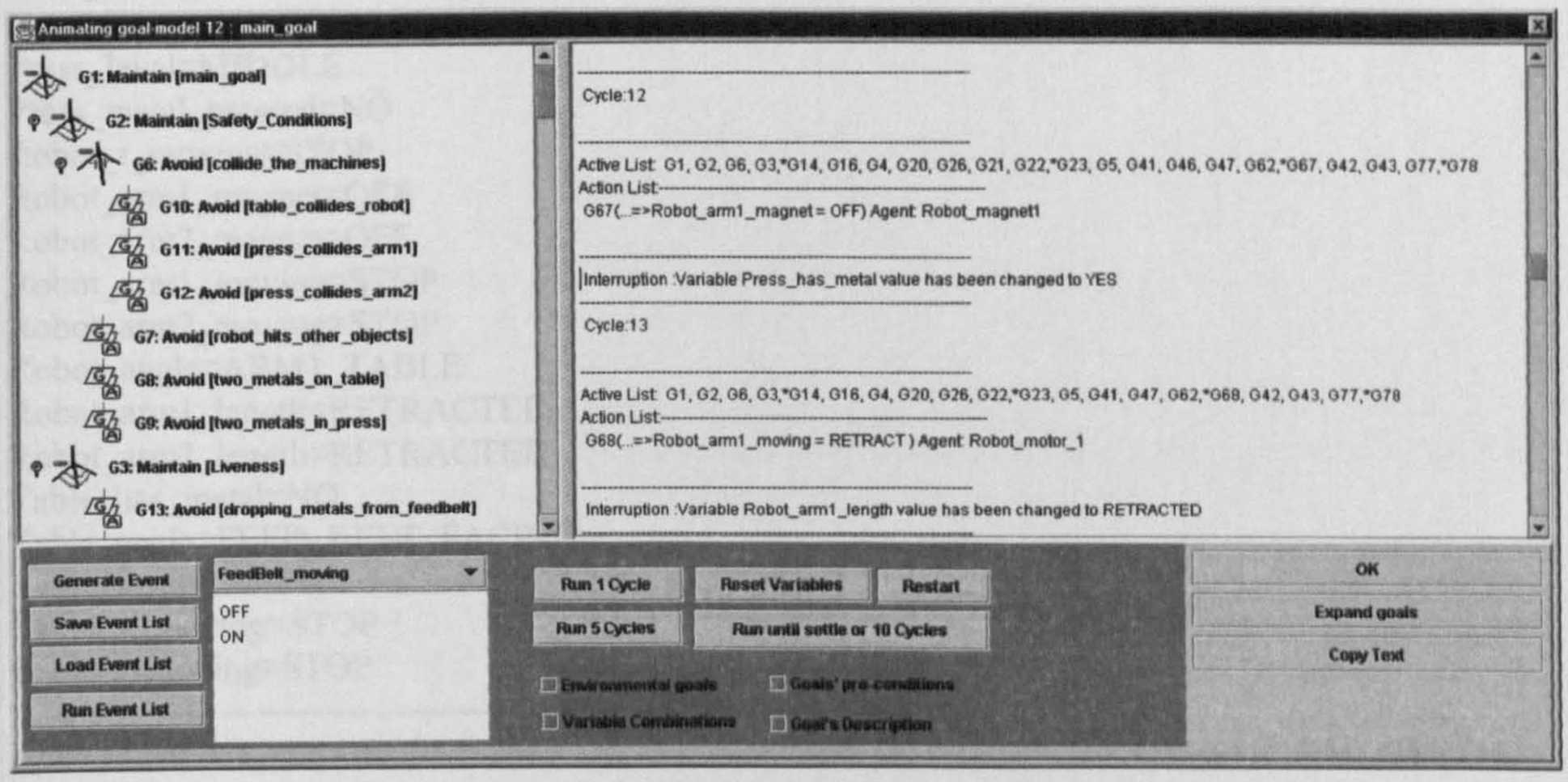


Figure 8.26, the animation utility of the GOPCD tool

An event list is a sequence of variable assignments (combination of variables and comparable values assigned to them); each of these variable assignments is associated with one execution cycle. This enables the user to specify multiple events in one cycle. The application is assumed to start from the initial situation at cycle 0; and at the start of each cycle, the relevant assignments will be executed. Table 8.6 shows an example for testing the production cell. This example starts from the moment that there is a metal on the table. Cycles may be skipped to simulate the physical time it takes to change a sensor reading; or to observe whether the system will settle or not (reach a state where there is no activated functional goals attempting to change the state).

Table 8.6, sample of an event list of the production cell application

Cycle	Variable name	Value
0	Table_has_metal	YES
1	Table_level	UP
3	Table_angle	ROBOT_FACING
4	Robot_arm1_length	EXTENDED
6	Table_has_metal	NO

The following list is the animation result when running the event sequence in table 8.6. In the displayed result, we have omitted the variable list as well as the pre-conditions of the activated

terminal goals for each cycle to save space. However, as can be noticed from the figure, the user can decide whether to see the variable list, pre-conditions and/or the environmental goals.

```
Animating goal-model 12 main_goal
 Resetting all Variables to the initial values:
-----------------------------------------------------------------------
FeedBelt_moving=ON
FeedBelt_metal_at_start=NO
FeedBelt_metal_at_end=NO
DepositBelt_moving=ON
DepositBelt_metal_at_start=NO
DepositBelt_metal_at_end=NO
Press_v_moving=STOP
Press_press_state=OFF
Press_has_metal=NO
Press_level=MIDDLE
Press_metal_pressed=NO
Robot_r_moving=STOP
Robot_arm1_magnet=OFF
Robot_arm2_magnet=OFF
Robot_arm1_moving=STOP
Robot_arm2_moving=STOP
Robot_angle=ARM1_TABLE
Robot_arm1_length=RETRACTED
Robot_arm2_length=RETRACTED
Table_has_metal=NO
Table_angle=FEED_BELT_FACING
Table_level=DOWN
Table_v_moving=STOP
Table_r_moving=STOP
-----------------------------------------------------------------------
Active List:  G1, G2, G6, G3,*G14, G16, G4, G20, G26, G21, G22,*G23, G5, G41,*G44, G47, G42,
G43,*G78
Action List:
 G44(...=>FeedBelt_moving = ON ) Agent: FeedBelt_motor
-----------------------------------------------------------------------
 Interruption :Variable Table_has_metal value has been changed to YES
-----------------------------------------------------------------------
 Cycle:0  State:...
-----------------------------------------------------------------------
Active List:  G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G48, G46,
G47, G42, G43,*G78
Action List:-----------------------------------------------------------
 G48(...=>Table_v_moving = UP ) Agent: Table_v_motor
-----------------------------------------------------------------------
 Interruption :Variable Table_level value has been changed to UP
-----------------------------------------------------------------------
 Cycle:1  State:...
-----------------------------------------------------------------------
Active List:  G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G49, G46,
G47, G42, G43,*G78
Action List:-----------------------------------------------------------
 G49(...=>Table_v_moving = STOP ) Agent: Table_v_motor
-----------------------------------------------------------------------
 Cycle:2  State:...
-----------------------------------------------------------------------
Active List:  G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G50, G46,
G47, G42, G43,*G78
Action List:-----------------------------------------------------------
```

```
G50(...=>Table_r_moving = RIGHT ) Agent: Table_r_motor
---------------------------------------------------------------
Interruption :Variable Table_angle value has been changed to ROBOT_FACING
---------------------------------------------------------------
Cycle:3  State: ...
---------------------------------------------------------------
Active List:  G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G51, G46,
G54,*G55, G47, G42, G43,*G78
Action List:----------------------------------------------------------
 G51(...=>Table_r_moving = STOP ) Agent: Table_r_motor
 G55(...=>Robot_arm1_moving = EXTEND ) Agent: Robot_motor_1
---------------------------------------------------------------
Interruption :Variable Robot_arm1_length value has been changed to EXTENDED
---------------------------------------------------------------
Cycle:4  State:...
---------------------------------------------------------------
Active List:  G1, G2, G6,*G10,*G7, G3,*G14, G16, G4, G20, G26, G21,*G23, G5, G41, G45, G46,
G54,*G56, G47, G42, G43,*G78
Action List:----------------------------------------------------------
 G10(...=>Table_r_moving = STOP ) Agent: Table_r_motor
 G7(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 G56(...=>Robot_arm1_moving = STOP ) Agent: Robot_motor_1
---------------------------------------------------------------
Cycle:5  State...
---------------------------------------------------------------
Active List:  G1, G2, G6,*G10,*G7, G3,*G14, G16, G4, G20, G26, G21,*G23, G5, G41, G45, G46,
G54,*G57, G47, G42, G43,*G78
Action List:----------------------------------------------------------
 G10(...=>Table_r_moving = STOP ) Agent: Table_r_motor
 G7(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 G57(...=>Robot_arm1_magnet = ON ) Agent: Robot_magnet1
---------------------------------------------------------------
Interruption :Variable Table_has_metal value has been changed to NO
---------------------------------------------------------------
Cycle:6  State:...
---------------------------------------------------------------
Active List:  G1, G2, G6,*G10,*G7, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G23, G5, G41,
G47, G42, G43,*G78
Action List:----------------------------------------------------------
 G10(...=>Table_r_moving = STOP ) Agent: Table_r_motor
 G7(...=>Robot_r_moving = STOP ) Agent: Robot_r_motor
 G17(...=>Robot_arm1_magnet = ON ) Agent: Robot_magnet1
```

The user can modify the requirements to remove logical errors after observing them through the animation. In appendix B section B.2.2, we provide an event list from the time the table received the blank metal until the robot delivers it at the start of the deposit belt. However, this animation utility allows the user to initialise the production cell at any configuration to test or observe particular behaviours.

8.4 Generating the Formal specifications

Unlike the previous phases, the third phase for generating the specifications will be hidden from the systems engineer; thus he/she does not have to know much about B or formal methods. However, the GOPCSD tool will attempt to produce documented formal specifications as much as possible using the informal descriptions of the goals, variables, agents and data types.

8.4.1 Generating formal operations/use-cases

The general operations specifying the Production cell will be generated from the terminal goals of the *main_goal* goal-model. The tool generates operations with pre- and post-conditions and an actor or agent who is capable of performing such operations. Each operational terminal goal (neither environmental nor non-functional) will be translated into an invariant. The generated operations are listed in table 8.7.

Table 8.7, the operations of the Production Cell

Invariant no. 1 table_collides_robot Actor(Agent): Table_r_motor
/* G10: table_collides_robot avoid table collides robot*/ **Pre-condition:** Table_level = UP and Table_angle = ROBOT_FACING and Robot_arm1_length = EXTENDED **Post-condition**: avoid table collides robot Table_r_moving = STOP
Invariant no. 2 press_collides_arm1 Actor(Agent): Press_motor
/* G11: press_collides_arm1 avoid the press tray hitting the robot first arm when it is in the middle position*/ **Pre-condition:** Robot_angle = ARM1_PRESS and Robot_arm1_length = EXTENDED and Press_level = MIDDLE and not (Press_v_moving = STOP) **Post-condition**: avoid the press tray hitting the robot first arm when it is in the middle position Press_v_moving = STOP
Invariant no. 32 move_robot_to_table Actor(Agent): Robot_r_motor
/* G52: move_robot_to_table move the robot so that arm1 faces the table*/ **Pre-condition:** (Table_level = UP and Table_has_metal = YES and Table_angle = ROBOT_FACING and not (Robot_r_moving = RIGHT) and not (Robot_angle = ARM1_TABLE) and Robot_arm2_length = RETRACTED and Robot_arm1_length = RETRACTED) and Table_has_metal = YES and Robot_arm1_magnet = OFF and Robot_arm2_magnet =OFF and Press_has_metal = NO **Post-condition:** move the robot so that arm1 faces the table Robot_r_moving = RIGHT
Invariant no. 45 ARM1_RETRACTS Actor(Agent): Robot_motor_1
/* G68: ARM1_RETRACTS arm1 is fully extended and holding the metal retract arm1 until it is fully retracted*/ **Pre-condition:** arm1 is fully extended and holding the metal (Robot_arm1_length = EXTENDED and Robot_arm1_magnet = OFF) and (Robot_angle = ARM1_PRESS and Press_has_metal =NO and Press_level = MIDDLE) and Press_has_metal= NO and Robot_arm1_magnet =ON and Robot_arm2_magnet = OFF and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP and Robot_arm1_magnet = ON **Post-condition:** retract arm1 until it is fully retracted Robot_arm1_moving = RETRACT
Invariant no. 56 Extend_arm2 Actor(Agent): Robot_motor_2
/* G84: Extend_arm2 arm2 is free and retracted Extend arm 2*/ **Pre-condition:** arm2 is free and retracted (Robot_arm2_length = RETRACTED and Robot_arm2_magnet = OFF and not (Robot_arm2_moving = EXTEND)) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and Robot_r_moving = STOP **Post-condition:** Extend arm 2 Robot_arm2_moving = EXTEND
Invariant no. 68 conveyProcessedMetal Actor(Agent): FeedBelt_motor
/* G78: conveyProcessedMetal always Convey Processed Metals to the collection area by keeping the deposit belt motor switched on.*/ **Pre-condition:** always **Environmental Assumption:** Convey Processed Metals to the collection area by keeping the deposit belt motor switched on.

In appendix B, section B.2.3, a full list of the operations is provided.

8.4.2 Generating B machines for the Production Cell

The GOPCSD tool translates the goal-model automatically to a B specification without user interaction. In this case study, there are ten agents controlling the press, robot, table and feed belt components. These agents control the production cell variables and maintain the safety and liveness conditions.

the generated B machines

Robot_arm2_moving_actuator.mch | Table_v_moving_actuator.mch | Table_r_moving_actuator.mch
Robot_arm1_magnet_actuator.mch | Robot_arm2_magnet_actuator.mch | Robot_arm1_moving_actuator.mch
Press_v_moving_actuator.mch | Press_press_state_actuator.mch | Robot_r_moving_actuator.mch
B machines list | datatypes.mch | maincontroller.mch | FeedBelt_moving_actuator.mch

OK

```
MACHINE  Robot_arm1_moving_actuator
SEES datatypes
VARIABLES
                Robot_arm1_moving       /* output variable */

INVARIANT
                Robot_arm1_moving : ARM_MOTION

INITIALISATION
                Robot_arm1_moving := STOP

OPERATIONS
set_Robot_arm1_moving_EXTEND =
PRE  Robot_arm1_moving /= EXTEND
THEN
Robot_arm1_moving := EXTEND
END;

set_Robot_arm1_moving_STOP =
PRE  Robot_arm1_moving /= STOP
THEN
Robot_arm1_moving := STOP
END;

set_Robot_arm1_moving_RETRACT =
PRE  Robot_arm1_moving /= RETRACT
THEN
Robot_arm1_moving := RETRACT
END
END
```

Figure 8.27, the Robot's first arm extendable motion of the Production cell

the generated B machines

Robot_arm2_moving_actuator.mch | Table_v_moving_actuator.mch | Table_r_moving_actuator.mch
Robot_arm1_magnet_actuator.mch | Robot_arm2_magnet_actuator.mch | Robot_arm1_moving_actuator.mch
Press_v_moving_actuator.mch | Press_press_state_actuator.mch | Robot_r_moving_actuator.mch
B machines list | datatypes.mch | maincontroller.mch | FeedBelt_moving_actuator.mch

OK

```
MACHINE  datatypes
SETS
ONOFF := {OFF, ON }             /* two values for valves */;
VERTICAL_MOTION := {UP, STOP, DOWN }    /* for motors then move up down or stop */;
PRESS_POSITION := {UP, MIDDLE, DOWN }   /* the positions of the tray inside the press */;
ROTATION := {RIGHT, STOP, LEFT }        /* three values for rotary motors */;
ARM_MOTION := {EXTEND, STOP, RETRACT }  /* for the robot arms control */;
ROBOT_POSITIONS := {ARM1_TABLE, ARM1_PRESS, ARM2_PRESS, ARM2_DEPOSITBELT }  /* */;
ARM_POSITION := {RETRACTED, EXTENDED } /* the robot arm positions */;
TABLE_R_POSITION := {FEED_BELT_FACING, ROBOT_FACING }           /* the angel of the table */;
TABLE_POSITION := {UP, DOWN }           /* the rotatry table vertical levels down aligned to the feed belt up aligned to the robot */
END
```

Figure 8.28, the data types of the Production cell

Since the deposit belt table is not controlled by the production cell, it would be useful to remove it from the list, as well as the two corresponding sensors; however, the user may leave the component itself to maintain the structural information about the production cell (as a part of the informal description of the system).

However, the GOPCSD tool carries out dependency and effectiveness analyses, to include only the controlled output variables and the effective input variables.

The GOPCSD tool generates a complete set of B machines to control the production cell. The generated machines have three types as follows:

- A Data Type machine that stores the definition of the enumerated data types used in the other machines.
- Actuator machines representing each agent and the operations that the agent can perform, such as rotate robot right or press the metal.
- A main controller machine that represents the entire goal-model; it includes the actuator machines. The controller machine has operations representing the agents' contribution to the entire goal-model; each of these operations has an IF_THEN_ELSE structure composed of the terminal goals grouped by the output variables.

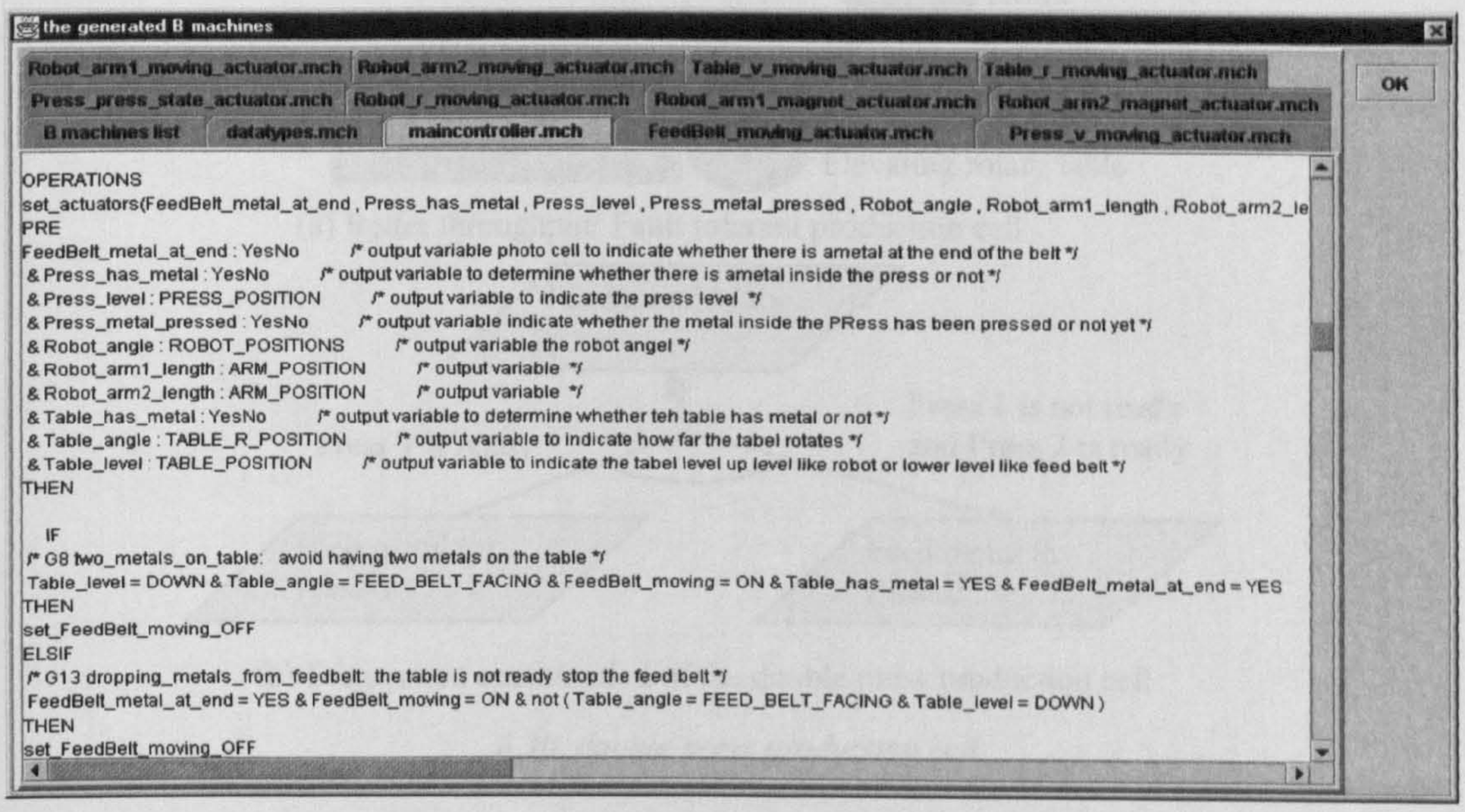


Figure 8.29, the Production cell B specifications, the main controller

Figure 8.29 shows a segment of the main controller machine that represents goal-model *main_goal*. The upper tabs of the dialogue box show the names of the different generated machines in addition to an index file "B machines list" that lists the B machines. As can be seen in the figure, the machine is documented as much as possible using the informal description of the GOPCSD tool entities. This can guide the software engineer to have considerable understanding of the application.

A full list of the generated B-machines for the simultaneous and sequential versions is given in appendix B, section B.2.3.

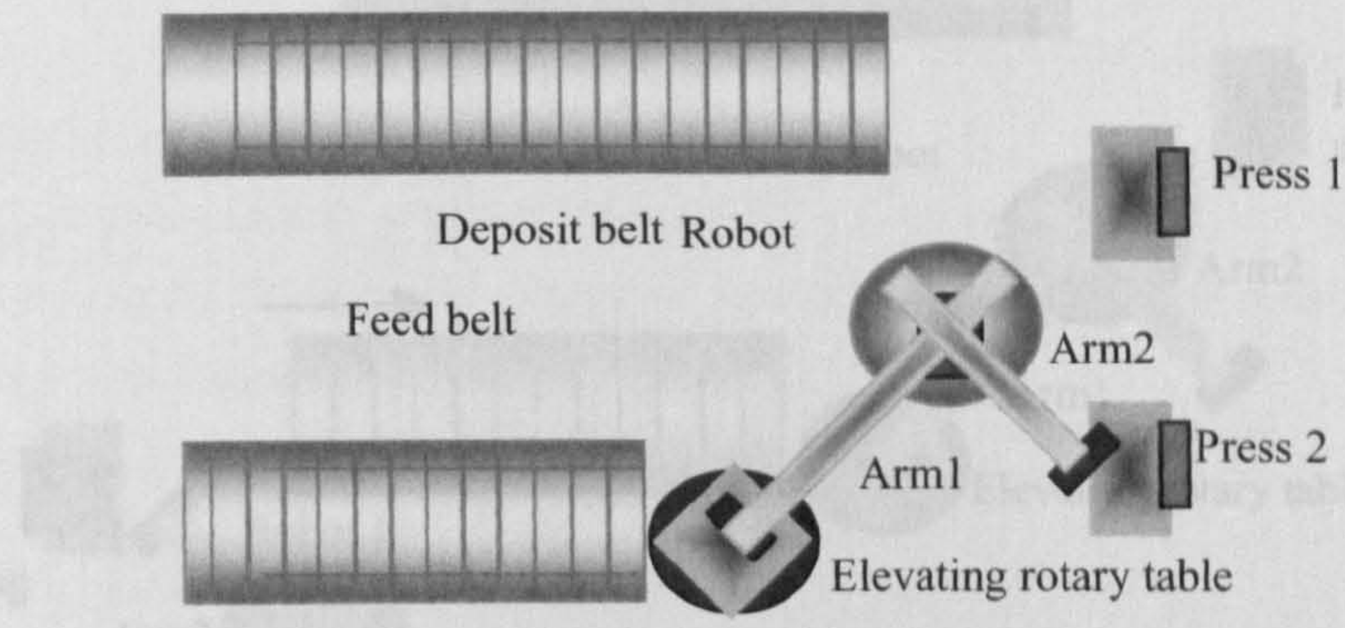
8.5 Extending the production cell

In this section, we illustrate how the production cell requirements can be extended in different situations, where the systems engineer will be able to reuse the simple production cell goal-model(s) and does not have to start from the beginning. We illustrate two situations as follows:

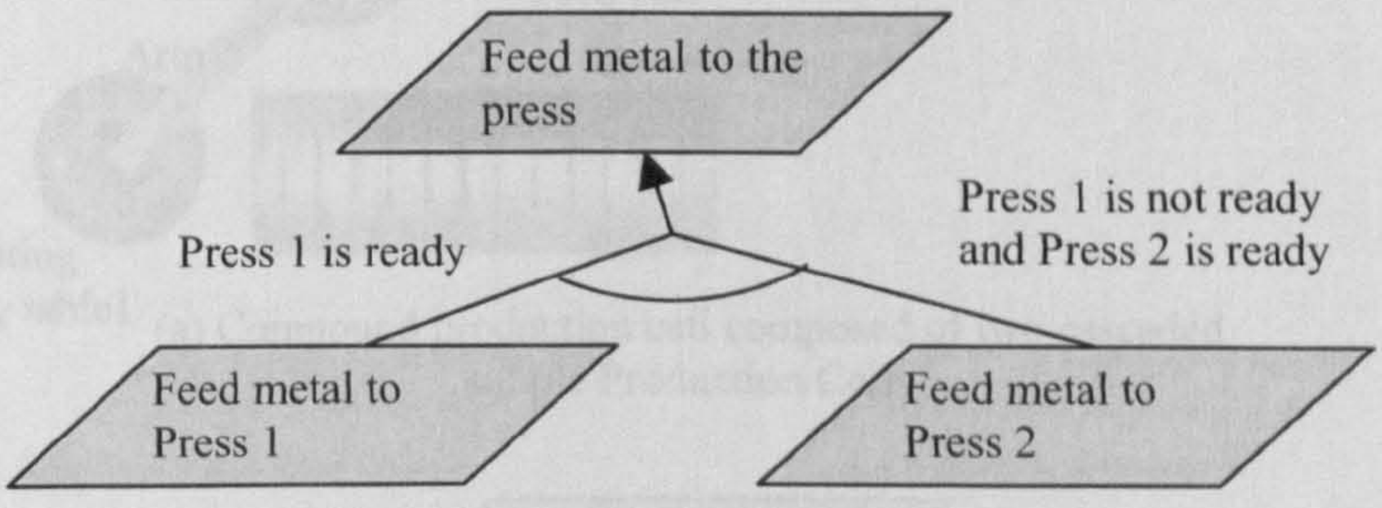
8.5.1 A Double Press Production Cell

To increase the throughput of the cell, the press can be duplicated to allow the robot to pick up a new metal and place it in a free press when the other press is busy.

The existence of a second press can be utilised similarly to tolerate any fault that can happen to the press by providing a stand by press that operates till putting the first press back to normal operation. In these two cases, the original requirements of a simple production cell can be reused, provided that it has been well structured.



(a) Better throughput/ Fault tolerant production cell



(b) Disjunction combination of the double press production cell

8.30, double press production cell

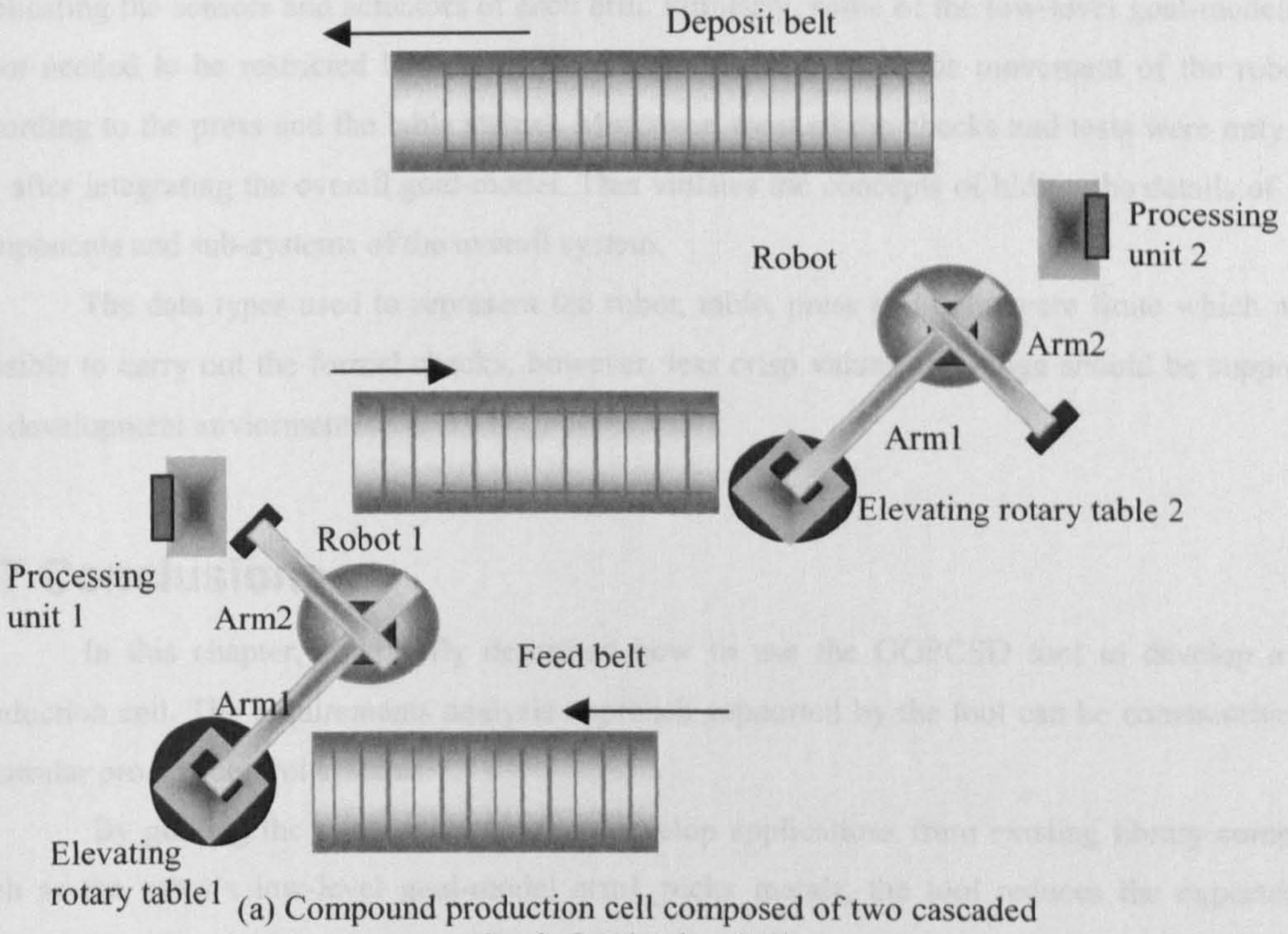
For example, in the main goal-model of section 8.4, most of the requirements goal-models that do not address the press can be reused. Whereas, those concern the press need to be duplicated one copy for each press. Finally, goals that involve the robot movement to the press should be modified. This can be accomplished easily by duplicating these goals (one instance for each press) and then creating a new parent goal that combines them using disjunction refinement pattern as shown in figure 8.30b.

8.5.2 A Cascaded Production Cell

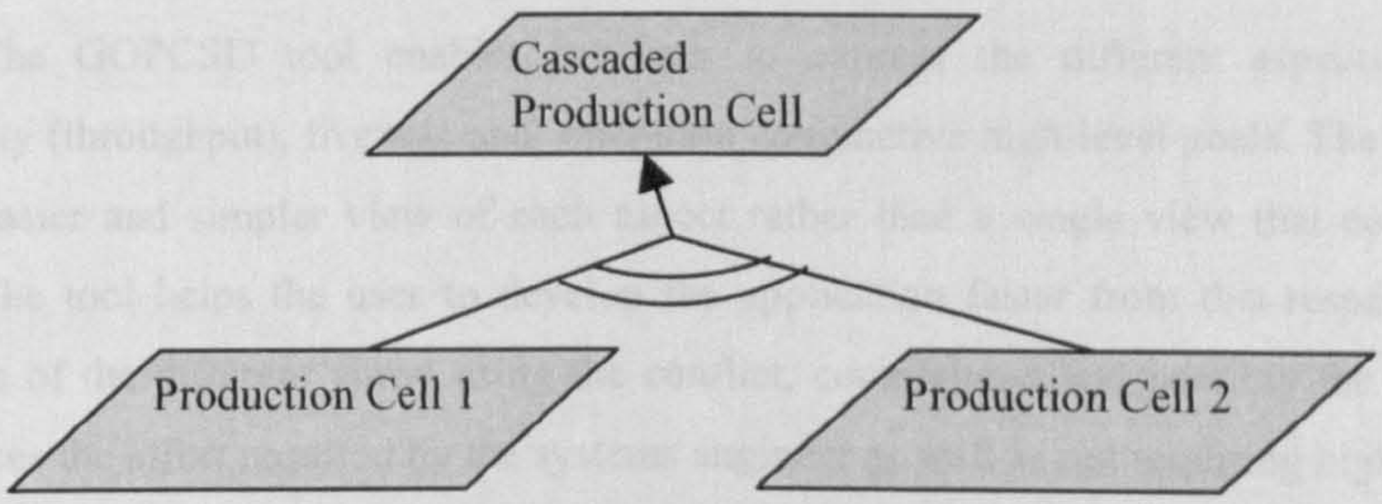
Another way to extened this simple production cell is to cascad another cell at its end as shown in figue 8.31a. This will enable the production line to process the metals or the products in multi stages. The requirments of the overall simple production cell can be now regarded as a component called "production cell". Therefore, the entire application can be considered as a compostion of two simpel production cells, where the deposit belt of the first cell srevs as a the feed belt for the second

one (this requires some mapping between the sub parts of this intermediate belt, which is already offered in the GOPCSD tool).

The operation of the new application can be accomplished simply by creating a higher goal that has two sub-goals each of them represents one of the simple production cells, as shown in figure 8.31b.



(a) Compound production cell composed of two cascaded simple Production Cells



(b) Conjunctive combination of the requirements of the two simple production cells to form the cascaded production cell

8.31, cascaded production cell

8.6 Disscusion

In this case study, we did not use the alternative refinement pattern to express any of the goals; however, this pattern can be used similar to the gas burner case study in chapter 7.

Moreover, the conflict and compelteness checks have not employeed in a "divide and conquer" fashion. For example, the sub-goal models had to be checked for conflicts then the overall goal model needed to be checked again as a whole. This increases the effort from the systems engineer side, and makes it difficult to eliminate the incompleteness and the inconsistency cases.

The checks should be carried in a fashion that groups the situations which reveals conflicts, incompleteness, indeteminancy, etc. rather than entailing the systems engineer to resolve each single instance of these cases.

Building components out of components and sub-systems were not fully supported by the tool. For example the robot was supposed to be built of two arms (as components) rather than duplicating the sensors and actuators of each arm. Similarly, some of the low-level goal-models of the robot needed to be restricted by adding pre-conditions (to control the movement of the robot arms according to the press and the table status). Moreover, most of the checks and tests were only carried out after integrating the overall goal-model. This violates the concepts of hiding the details of internal components and sub-systems of the overall system.

The data types used to represent the robot, table, press and belts were finite which makes it possible to carry out the formal checks; however, less crisp values and range should be supported by the development enviorment.

8.7 Conclusions

In this chapter, we briefly described how to use the GOPCSD tool to develop a simple production cell. The requirements analysis approach supported by the tool can be constrcutively used in similar process control systems.

By guiding the systems engineer to develop applications from existing library components, such as the robot's low-level goal-model arm1_picks_metals, the tool reduces the expected effort expended by the systems engineer. This enables the systems engineer to focus on the intgration of these components rather than defining them.

The GOPCSD tool enables the user to express the different aspects, such as safety, productivity (throughput), liveness and operation conjunctive high-level goals. The user may possibly have an easier and simpler view of each aspect rather than a single view that combines these four aspects. The tool helps the user to develop the application faster from this respect; it provides the integration of the different views using the conflict, completenss and possibly the animation utlities. This reduces the effort required by the systems engineer as well as not requiring high-level expertise in the process control field. The way the different aspects are separated makes it is easy for the user to deicde which goal to weaken or strengthen when resolving conlficts [El-Maddah and Maibaum 04b]; in addition, it helps to increase the modifabilty, traceability, and augmentability [Davis 93] of the entire production cell.

Applying the obstacle analysis to throughput goals enabled the systems engineer to realise that the goal is obstructed by non-avoidable obstacles; and hence, he/she had to de-idealise the goal (modify the requirements into more realistic ones).

The modification decisions were addressed to the "right" person, the systems engineer, rather than waiting for the formal method feedback. Otherwise, the software engineer's feedback which is based on animating or verifying the formal specifications, demands from him/her to correct the logical errors of the requirements, something which is considered to be outside his/her scope.

Conclusions and Future Work 9

In this chapter, we draw the main conclusions from the research and compare the principles used in developing applications with the GOPCSD tool with those of other methods. Also, we provide some suggestions for future work and possible extensions of the GOPCSD tool.

9.1 Conclusions

As noted in [Pressman 92, Davis 93, Nuseibeh and Easterbrook Y2K, Gilb 03], the more attention paid to the requirements stage, the more constructive, more guided, and less exhausting will be the design, implementation and testing stages. And hence, the software application will have a higher chance of achieving success. Leaving logical errors at the requirements stage will lead to an accumulation of errors during the design and implementation (i.e. result in building an erroneous design and implementation [Davis 93]).

We have proposed a detailed method and supporting tool to develop complete, consistent, reachable and traceable requirement specifications for process control systems. The main objective was to separate the requirements and the specification and design concerns, and hence to guide the systems engineer in structuring and refining the user needs, even though they may initially exhibit incompleteness, inconsistency and/or logical errors. The GOPCSD tool was shown to be considerably helpful when performing the various checks, validating the requirements and generating the B specifications. We highlight these aspects, as follows:

I. Adopting the goal-oriented approach to analyze the requirements, as in [van Lamsweerde et al. 91, Liu and Yu 91, Antón 95, Mylopoulos Y2K, Heaven and Finkelstein 03], has a constructive impact in utilizing the "*divide and conquer*" concept to reduce the effort required by the user. The effort has been split into the construction of the low-level goal-models and the high-level goal models, which are already offered by the tool library, and integration (refining and combining the separate goal-models), which is left for the tool user. This advantage is not specific to GOPCSD, but general for goal-driven methods.

II. Shortening the time and reducing the effort required to develop the complete object-model as compared to the general KAOS method can be considered a practical gain, especially in applications such as process control systems, where there is some minimum level of understanding of the system to be developed and of the environment. Moreover, the systems engineer is able to express the system's states simply as a list of variables [El-Maddah and Maibaum 03a]. This reduction in effort enables the systems engineer to obtain a time saving from the tool, particularly in medium-scale systems, where the user expects the tool to shorten the development time and reduce the effort, as shown in the production cell case study, where the systems engineer did not need to describe the relationships between the robot, blank metals, etc. In addition, the implied relationships of the object-model as explained earlier in chapter 5 enable the systems engineer to focus on the system's operation rather than on the unimportant relationships between the environment components or even the sub-components of the systems, which often, requires more effort and talent in modelling the system and the environment using UML diagrams.

III. A goal-model can be considered as a more expressive means of describing requirements compared to a STD, state chart or R-net, due to the fact that each goal-model starts with an abstract goal, which is refined into formal and functional terminal sub-goals; this enables the normal systems engineers to use the tool with good understanding, even though they do not have high expertise in the mathematics or logic underlying the use of STDs, state charts or event response diagrams. Although the state-charts have a hierarchical form, the goal-model is more easily understood, especially in the cases where the number of refinement levels is large. It is yet still easier to be understood, modified, evolved, extended, reused, traced to the implantation/design stage and traced from the user needs. Again, this advantage is not specific to GOPCSD, but general for goal-driven methods.

IV. The goal-model's goals can be refined to functional terminal goals and/or assumptions about the system and the environment. These assumptions will be placed in the appropriate sites within the goal-model where they can supplement with the functional sub-goals to express higher-level goals. This can be helpful in reporting the different assumptions required in different situations.

V. Ability to trace the different requirement aspects like safety, liveness, and enhancing the application performance; furthermore, the user can start the requirement development with a local view of each aspect and less awareness of other aspects; then the tool can check the consistency of the overall application. This encourages building aspect goal-models for safety and liveness in the GOPCSD library to use them in similar applications.

VI. Building applications from components as in [Leveson et al. 94] is considered as a constructive aspect of the GOPCSD tool, as in [El-Maddah and Maibaum 04c]; it enables the "*divide and conquer*" concept as well, to focus on the component at one level and then focus on the interactions between the different components at a higher-level. Moreover, the hierarchal structure of the requirements and constructing the application from components increase the extendability, maintainability, and augmentability of the applications' requirements.

VII. The identified goal refinement patterns were shown to be helpful in guiding the systems engineer to structure the requirements at both the low- and high-levels. In addition, these refinement patterns, which are considered as specialized temporal/logical cases of the general KAOS AND and OR patterns, provide considerable guidance to the user in reasoning about and debugging the constructed requirements since they are closer to his/her perspective level and address rational concepts like sequentiality, alternation and simultaneity.

VIII. The alternative refinement pattern enables the user to express different solutions combined together in a single compact goal-model. The requirement specifications shared between the different solutions will be represented only once, until the user wants to generate the separate solutions. He/she can use the split utility to generate the simple solutions.

IX. The disjunction refinement pattern [El-Maddah and Maibaum 03a] has an expressive power in situations where the main goal needs to be differently fulfilled from one situation to another during run-time of the application. This pattern is difficult to express using the normal KAOS AND/OR patterns, as explained before in chapter 5, this pattern can effectively express situations where different arrangement can be used to fulfill one goal during the operation based on the entire application state.

X. The second phase of the GOPCSD tool, which involves the various checks and tests for the requirements, enables early correction of requirement errors. This correction will be assigned to the appropriate expert, the systems engineer. He/She will be responsible for taking the requirement-modification decisions rather than the software engineer; this can reduce the potential interaction between the systems and software engineers and increase the separation of the concerns between them.

XI. The GOPCSD tool enables the user to reason about goals and trace the variable/component inter-dependencies; this is shown to be important, especially in understanding large-scale systems and having conceptual decomposition, which is considered as an advantage to increase understandability and analysability for the systems engineer and later for the software engineer.

9.2 Comparison with related methods

Categorizing the GOPCSD tool as goal driven-utility requirements analysis forces us to compare our work with the KAOS method [Letier and van Lamsweerde 02] since it deals with functional requirements; on the other hand, generating B specifications from the tool allows us to compare our work with other reactive systems design methods and tools such as [Lano et al. Y2Ka, Sekerinski 98].

Firstly, we consider the related work of developing B specifications; as explained earlier in chapter 2, Sekerinski in [Sekerinski 98] attempts to use state transition diagrams and statecharts as the first user interaction element; this should be helpful provided that there is a good supporting tool, the user provides correct design of the states and their transitions, which may be difficult to ensure, especially in large scale systems with compound interaction between the sub-systems.

On the other hand, in [Lano et al. Y2Ka] (chapter 2), Lano et al. have built a tool (RSDS) that supports the automatic generation of B specifications for reactive systems. The principle differences in the developed specifications between the different approaches can be summarized as follows:

I. The requirements in the GOPCSD have a goal-oriented nature. This makes the GOPCSD tool have a capability of breaking down the complex invariants/operations; thus, instead of starting with complex invariants/operations at high-level goals, they can be broken down to simpler ones at the sub-goals levels. On the other hand, the RSDS user is required to enter such complex invariants/operations. And similarly, in the graphical design method state charts of the same level of complexity have to be designed.

II. In GOPCSD, there is a concept of systems and sub-systems and building the systems out of their components; on the other hand parallel states are allocated to the sub-systems in the graphical design method, and there are concepts of controller, sensor and actuator in RSDS.

III. Traceability is very clear in the GOPCSD tool where the high-level goals represent aspects, like safety, liveness, economy and throughput while the user needs apart from safety in the RSDS tool are scattered within the entered invariants. Furthermore, tracing the requirements to the design level is supported in the GOPCSD through adding comments (including the goals names and informal descriptions) on the generated B machines. But this is not the case in the graphical method of design reactive system or RSDS.

IV. There is no support for sequence patterns in RSDS while in the GOPCSD tool the user can define sequences of goals and the tool ensures that each goal will be considered as a pre-condition to its successor goal.

V. In the RSDS tool, the user gets involved with state charts, state transitions, data control flow diagrams and UML class diagrams. Which may be considered difficult for a systems engineer to cope with. Whereas, in GOPCSD the user is supposed only to understand the goal-models.

VI. The application development lifecycle starts by providing the user with low-level goal-model specifications of the components and high-level templates of the application from the provided GOPCSD library, whereas in the RSDS tool, there is no definition for a requirements /specifications library apart from some two states, on/off, sensors or actuators. This is considered as extra modeling effort for the systems engineer to perform.

VII. The incompleteness and inconsistency checks are applied in RSDS and GOPCSD tools. On the one hand in the GOPCSD, there is some guidance for the user to correct the conflicting goals or to complete the requirements; on the other, in the RSDS tool, the user is assumed to know how to correct the invariants. In addition, in the GOPCSD tool, we assume the requirements can prescribe logically erroneous behavior and provide the validation stage to

enable the user debugging the requirements; this concept is not supported in RSDS, the model checking is achieved through manually translating the RSDS format into SMV with difficulty to trace back to the suspected invariants.

VIII. The GOPCSD tool supports reachability check to test the possibility of reaching the states corresponding to terminal goals, which is not provided in the RSDS tool.

IX. The description of the requirements/specifications building units (the informal description of the goal in GOPCSD) works in conjunction with the formal description in the GOPCSD tool, while the RSDS is solely based on the formal definition, which is difficult to be understood for the non-logic/mathematical user.

X. Traversing the goal-model tree is considered as an easier way to understand the application (reasoning why and how), while this is not the case in RSDS where the requirement specifications are represented as a collection of parallel logical formulae.

XI. The GOPCSD tool generates B specification and general format operations (composed of pre- and post- conditions); the RSDS generates various output formats, such as VDM, PLC, B and JAVA.

XII. The working environment in the GOPCSD tool enables the user to attempt a number of solutions together; whereas in the RSDS tool, only single solution is supported at the time. Moreover, the ease and flexibility of changing some parts the solution need considerable effort compared to the GOPCSD tool.

Secondly, we consider [Letier and van Lamsweerde 02b] which uses similar goal-oriented requirements to our work, but has wider scope that covers software applications in general. Regarding the first stated advantage (chapter 3), with respect to the nature of process control systems, the level of abstraction is considered higher than the user's expectations at this stage; he/she expects to see some outlines of the implementation programs, stating the required variables, variable initialization, invariants, in addition to assignment statements and/or decision structures such as *IF*, *CASE*. With respect to the second advantage, the completeness check is already performed at the second phase within the GOPCSD. The fourth advantage is also considered irrelevant with respect to the GOPCSD method because the goal-model is assumed complete before starting the translation in the third phase.

On the other hand, one must consider the general scope covered in KAOS [Letier and Van Lamsweerde 02b] that is wider than the scope covered by the GOPCSD tool, where variables and changing their values are considered as being simpler, which enables the GOPCSD tool to formalize and automate the checks applied to the goal model and generate specifications nearer to the implementation level.

Regarding the above comparison, we can draw the following conclusions:

- From the Systems Engineer's perspective, the GOPCSD is considerably nearer to the systems engineer's mode of thinking (by adopting goal driven concepts, as in KAOS) than [Lano et al. Y2Ka] and [Sekerinski 98] and hides the B formal method's details.

- The generated specifications as B machines can be considered closer to the user expectations than [Letier and van Lamsweerde 02b]. This is because the state machines are an intermediate representation between the goal-model and the B specification, and from the latter there is a well-understood and formal path to generating executable code. In contrast, there is no corresponding implementation path described in [Letier and Van Lamsweerde 02b] providing the same levels of confidence to the systems engineer.

9.3 Contributions

Referring back to the objectives of the research mentioned in chapter 1 and having concluded and compared our research to other methods, we could argue that:

- Identifying the specialized refinement patterns used in process control applications guided the goal-oriented requirements construction and generation of B specifications.
- The GOPCSD supports reusability at the early requirements stage and builds component-based requirements.
- The tool allows the systems engineer to gradually bridge the gap between the user needs and the formal specifications.
- The GOPCSD tool potentially reduces the concern-interference between the systems and software engineers.
- The tool builds a precise and traceable requirements model that can be used later to evolve and maintain the initial requirements to enhance the developed application.

9.4 Limitations

We can briefly summarise some weaknesses of our work, as follows:

I. Whereas the concept of "*divide and conquer*" is used to construct the goal-model, it is not as well utilised in checking consistency, reachability and completeness of goal-models.
II. The functional terminal goals are restricted to assign deterministic values to the output variables. This enables the GOPCSD tool to perform the conflict, inconsistency and validation analyses that would otherwise be difficult.
III. Compared to [Lano et al. Y2Kb], the structure of the generated operations and the B machines is flat; this flat design is not normally recommended, especially in large–scale systems.
IV. The formalisation does not use the "for all", ∀, nor "there exists", ∃, logical operators. This restricts the goals to specify fixed structures of process control applications, as well as preventing the chance of describing non-determinism (e.g. using the "there exists", ∃, operator to describe one agent/variable from a number of agents/variables).
V. The formalisation stops at simple variables and their assigned enumerated [G1]values expressed as pre- and post- conditions. Although having simple conditional assignments makes it easy to check

consistency and completeness, it restricts the specification of the behaviour of the process application. (For example, one cannot specify that an output variable will increase or decrease.)

VI. GOPCSD does not support general parameterised predicates, as is the case in KAOS. The predicates can be considered as an intermediate level between the informal description and the formal specification of the system. For example, Robot_Arm(arm1, extended) can be used to indicate that the arm1 of the robot is extended (implicitly defining arm1, arm2 and their states (retracted, extended) as parameters to the Robot_Arm predicate); this centralises the formal definitions rather than scattering them among the different variables.

9.5 Future Work

After revising the contributions and the limitations of the research, we could focus the future work on two areas: extending the research and enhancing the GOPCSD tool. We recommend that further research is required in the following directions:

- Focusing on a sub-family of Process Control Systems like production cells or automotive applications should increase the chance of identifying new refinement patterns and building a requirements library of more specialized low-level components and high-level templates.
- Providing a general goal-model library of fault-tolerance analysis [Storey 94] should reduce the effort required by the systems engineer to carry out the various risk and fault tolerance analyses.
- Specifying existing B machines as goal-models and reusing them to build super applications; this reverse engineering aspect is very crucial, especially when augmenting an existing software application without documented requirements or specifications.
- Software agents could constructively accomplish various tasks like dispatching and scheduling.

On the other hand, the following steps could be considered beneficial for augmenting the GOPCSD tool:

- Structuring the B machine controllers, in order to produce easier machines to debug and use within the B toolkit environment.
- Proving in "temporal logic" that the built in state-transition diagrams can fulfil the application properties, provided that the properties are expressed separately from the goal definitions.
- Hardwiring other refinement types, such as pipelines, critical sections and recursive patterns; or, alternatively, enabling the senior systems engineer to create new patterns.
- Translating the goal-models to other formal languages, such as VDM++ and SMV or directly to high-level languages, like PLC or Java, either as complete code or interface classes. In addition, translating the goals or the non-functional aspects into properties that can be checked or proved at the specification level.

- Saving scenarios in a library to be used as a test bench for the developed goal-models, and allowing some basic evaluation concepts (interactively with the user to judge the different versions).
- In the case of removing the inconsistency between the goals, the user should be able to choose between modifying the goals' pre-conditions and prioritising them. The priority approach should help when evolving the conflict goals without changing their pre-conditions.
- Enhancing the graphical user interface (GUI) of the tool:
    - Having a visual schematic diagram of the component-wise structure of the application, in addition to a collapsing and expanding facility to show and hide the composite component's details.
    - Representing the goal-model graphically in a vertical orientation and allowing expandability and hiding of some levels.

Appendix A

The GOPCSD tool Documentation

A.1 Introduction

Over the last two decades, much effort has been put into automating the reactive systems [Wieringa 01] software industry. The automation generally decreases the time required to produce the application and enables users with less knowledge of programming and logic to build software applications. On the other hand, it acts to lower deviation from the main requirements and acts to avoid hazards during run-time for the software programs. This automation stage can start at requirements gathering, design, or implementation stages, or it can cover more than one stage. For example, with most formal methods [Hinchey 95], the automation starts from the specification level and covers both the design and implementation stages. Some other efforts address preliminary stage to that accepts formal specifications in some other form. This is then translated into some other form as a preliminary to applying more conventional formal methods.

A.1.1 The GOPCSD Tool Scope

Software tools are often built for specific families of applications. This enables the tool to be more powerful, sophisticated and aware of the special requirements of the applications family. Moreover, focusing on a narrow family of applications increases the chance of reusing both the hardware devices and software components in similar applications. Furthermore, reusing the same methodologies of design and requirements gathering can be more specific for such a narrow family. Our research focuses on Process Control Systems such as burner systems, hydraulic systems, production cells, and lift systems. Process Control Systems are Reactive Systems that communicate with their local environment through sensors and actuators. Such communication is continual: the system senses a number of variables and then reacts to them by producing one or more actions that change its local environment.

A.1.2 Why the GOPCSD tool is created

Building a software controller for a process control system generally needs two different types of knowledge: knowledge about the application and its operations, in addition to knowledge about programming language paradigms. Usually, the integration between these two types of knowledge is carried out through cooperation between the systems engineer in charge of the process control system and the software engineer in charge of creating the software controller. The systems engineer, who is supposed to have complete information about the operations of the process control

system, states the application requirements and then passes them to the software engineer who has sufficient background about Process Control Systems. These requirements are then processed by the software engineer and then translated to specifications or design documents: in case of specifications, the software engineer could use formal methods Tools (like the B-Toolkit [Wordworth 96, Lano 96]) that have the capability of translating the specifications into running versions that can control the Process Control System, or as another alternative, the software engineer writes a final version that satisfies the design in a programming language.

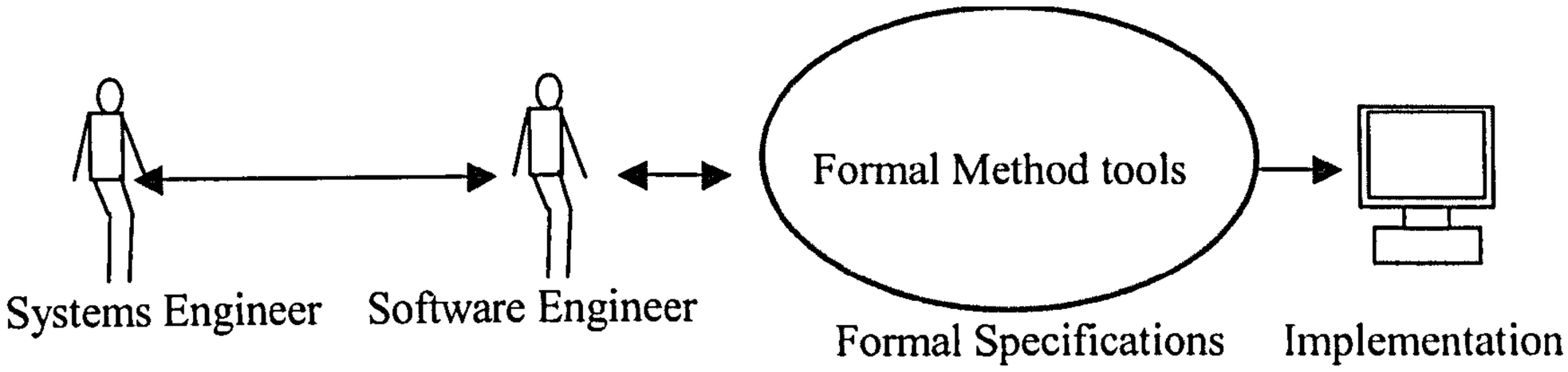


Figure A.1, the interaction between the system engineer and the Formal Tools

Mistranslation of the systems engineer's requirements can take place for different reasons: He may provide different levels of operational requirements that conflict with each other, that can produce inconsistency and/or ambiguity that has to be resolved by the software engineer, who knows less about the operations of the application. Besides, the systems engineer may find difficulties in expressing the details of the operations in a hierarchy of sequences and events that is needed by the software engineer. Some parts of the requirements can be missed out; for example under specific combinations of sensor readings what the appropriate actions to be taken. This can lead to incompleteness in the requirements that may result in hazards during the operation of the system. In addition to the correctness of the application requirements, in many cases the systems engineer is not fully aware of the logical errors that he made in the requirements. These logical/requirements errors could be detected through a validation stage in the formal method or later when the implementation version will be tested. In [Davis 93], it has been argued that the earlier the detection and removing of the requirements errors the easier, faster and cheaper the development.

Figure A.1 is a schematic diagram connecting the systems engineer and the implementation version of the control program. There is a distance between the systems engineer and the formal methods specifications that has to be filled through potentially co-operation between the systems engineer and the software engineer.

A.1.3 The requirements format within the GOPCSD tool

As mentioned above, there is an existing gap between the systems engineer's requirements and the formal specifications. One of the main objectives of this research is to reduce this gap through a tool that accepts the application requirements and guides the systems engineer to structure them in a format that eases both the completion of the requirements themselves and translating them into B formal specifications [Davis 93]. Such a tool should be suitable for the family of applications constituting process control systems, whose operations are more likely and more easily expressed as combinations of alternative/ successive/ parallel operations rather than as segments of a program as in

formal method specifications. Thus, a suitable structure for the requirements will be grouping the sub-operations as parallel, successive or alternative operations. The same idea can be applied at higher-levels of abstraction through grouping the compound operations together until reaching the highest-level function of the application that can be labelled as "*proper operations*" or "*safe run*". This idea can combine the operational requirements with the safety and economical requirements, rather than having a different design or view for each. Such relations between the different operations and such a hierarchy are already built in to the KAOS method "Knowledge Acquisition in Automated Specifications" [VanLamsweerde, 91].

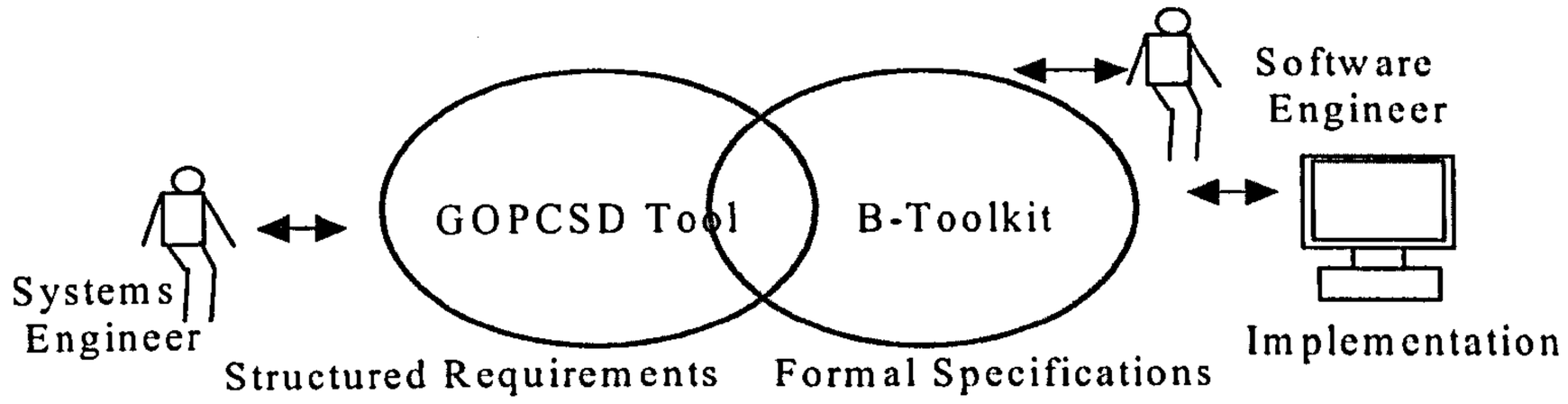


Figure A.2, the interaction between the system engineer and the GOPCSD Tool

KAOS is a general method to specify the intended application operations in terms of a hierarchy of goals. The goals are structured within a goal tree that has a single root (main goal). The main goal is refined into a number of sub-goals that have more details and a narrower view than the main goal. These refined goals may be further refined to sub-goals. When each refined goal is simple enough to be directly accomplished, it is called a terminal goal and it is associated with an agent. Thus, agents are responsible for accomplishing the terminal goals. They can be humans, devices like actuators or software components, either existing or to be produced later (e.g. dispatching or scheduling programs) [Letier 01].

A.1.4 How should the GOPCSD tool work

Thus, the Goal–Oriented Process Control System Design (GOPCSD) tool [El-Maddah, 03] is to implement an extension of KAOS called (KAOS+) in gathering the requirements from the user and then translating them into B specifications. Figure A.2 indicates the role of the GOPCSD tool between the systems engineer and the formal method (B toolkit). By use of the GOPCSD tool implementing the KAOS+ method, the application requirements will have hierarchical structure that can be shown to be helpful, as we will explain later. Such a hierarchy starts from the most abstract goal that is the highest–level goal and then moves down via refinements and more local views for the different parts, components, or operational modes.

Over and above filling the gap between the systems engineer and the formal specification, the GOPCSD tool is to assist its user in three main features: the first is to check the requirements using reachability, goal-conflict, completeness, and obstacle analyses, thus, removing early any bugs that may be detected later but with higher cost; the second feature is to enable the user to validate the system by animating the goal-model, starting from initial values for the system variables and allowing the user to simulate events and produce sudden changes in the output variables and observe the

application's response; the third feature is the reusability that has considerable importance, particularly when a process control system needs maintenance because of extending it or replacing a component with another one which is not identically the same. This extension or replacement imply that the process control system needs to be modified, which requires the systems engineer to restate the requirements again or reuse the previously stated ones.

Providing a library of the most common process control applications and components and its operations, the system engineer will be more likely to reuse the existing requirements to design the system with less effort and in a shorter time.

The GOPCSD tool's user is considered to have good knowledge of the system's physical construction and knows the different variables and agents that are related to the system. The user starts constructing the application requirements by importing the components, which include goal-models, agents and variables from the library or creating these entities within the tool. Then, the user defines the main goals of the application. These main goals can be defined in separate goal-models that will serve as workspaces; afterwards they have to be combined together to form a single goal-model. Within each main goal's sub-tree, the user can import goal templates from the library or refine the goals and perform some analyses and checks to ensure the intermediate correctness of the application.

With respect to the formality of the goal-models, the GOPCSD tool allows the user to informally describe the higher-level goals, and formally describe the lower-level goals. The user has to fully describe formally the terminal goals, in order to be able to produce B specification machines and perform the other checks like conflict checking, obstacle detections and animation.

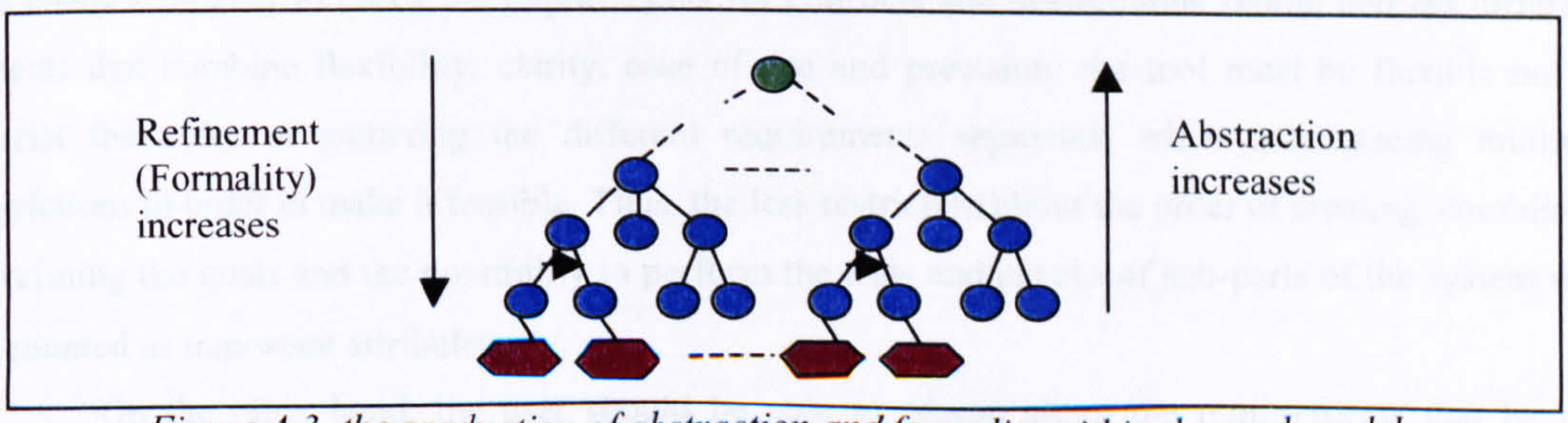


Figure A.3, the graduation of abstraction and formality within the goal-model.

This graduation in applying the formality within the goal-model splits the effort required to design the goal-model into multiple stages; in each of these stage the design will be easier and more straightforward. It should be obvious that, between the each level of the goal-model, the abstraction increases upward, while the refinement increases downward as shown in the figure A.3. After reaching the level of terminal goals, each terminal goal has to be associated with an agent and then be formally specified. By reaching such a state for each terminal goal and provided that there is only one goal-model (i.e. the initial separated goal-models are now combined into a single goal-model), the application is almost ready to be translated into B Specifications; however, it has to be checked. Thus, to check the correctness of the application the user should be able to perform consistency and completeness checks, moreover, obstacle analysis can be carried out at this level. The user can then proceed to validate the requirements through animating the goal-model, and finally, generate the equivalent B specification machines. The following sections provide the details of the tool design.

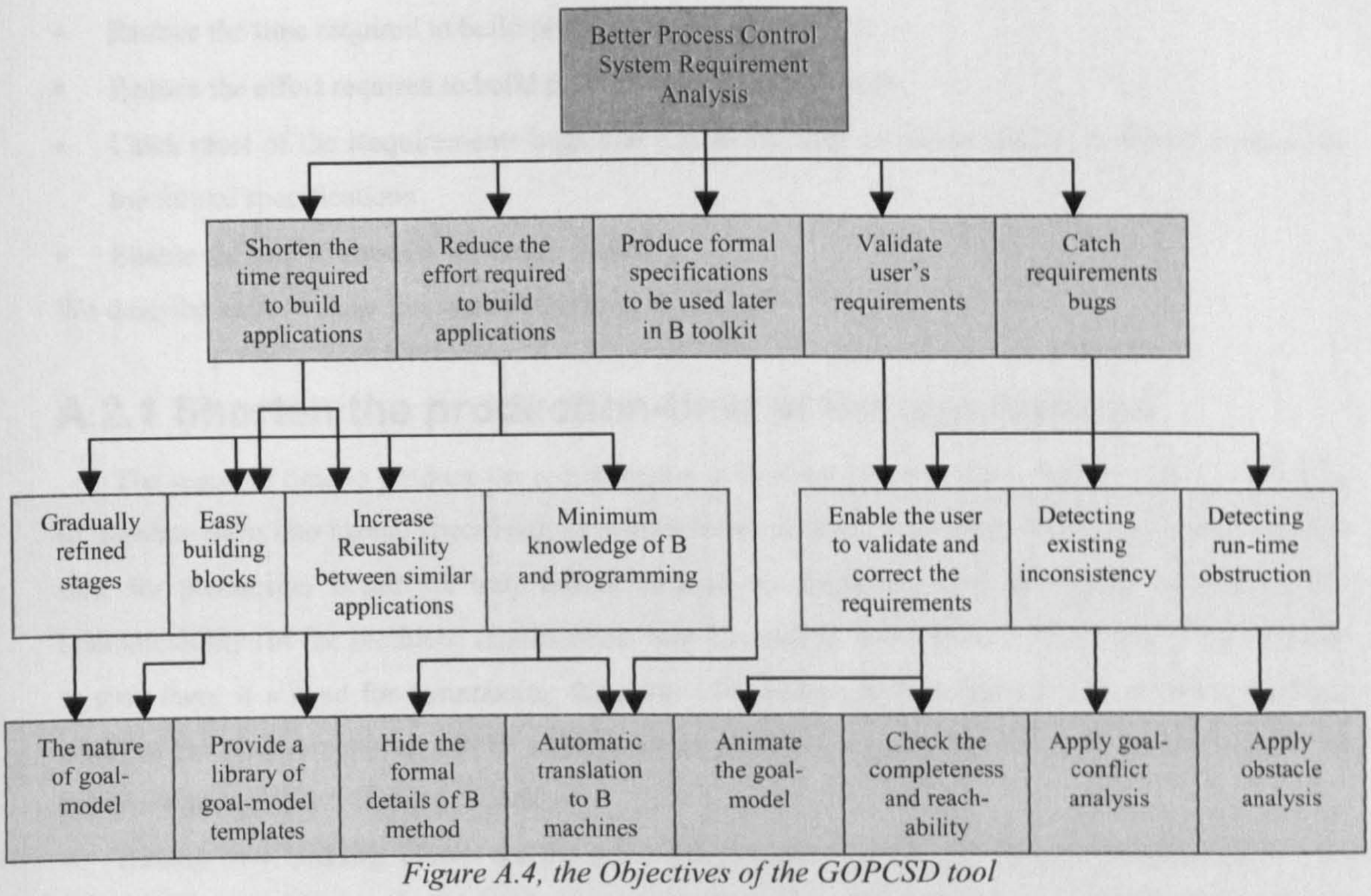


Figure A.4, the Objectives of the GOPCSD tool

A.2 General Objectives Required in the GOPCSD tool

Building a tool that automates the requirement analysis, interacts with the systems engineer, and enables him/her to check the requirements for conflicts and obstructions should address different aspects that combine flexibility, clarity, ease of use and precision: the tool must be flexible not to restrict the order of gathering the different requirements segments, while guaranteeing minimal restrictions in order to make it feasible. Thus, the less restriction about the order of creating, combining or refining the goals and the possibility to perform the tests and checks of sub-parts of the system will be counted as important attributes.

On the other hand, the user should be able to reason about the requirements that he has previously entered at the different levels; it can be of considerable aid as well for the systems engineer to be able to perform different checks on the goal-model to ensure the correctness of the requirements as early as possible. The tool must be precise in translating the logical and temporal hierarchy of goals and provide one way for the user to validate the requirements in order to ensure his /her agreement on the requirements before translating them into formal specifications.

The chart in figure A.4 represents a schematic diagram of the required objectives from such a tool. The main objective appears at the top; the main objective is then refined to more detailed sub-objectives until we reach the lowest–level objectives that are implemented within the GOPCSD tool. In the following text we describe the objectives in figure A.4, starting from top to bottom and left to right.

The highest–level objective is to build a better analysis tool for process control systems; we define "better" as possessing the following five characteristics:

- Produce formal specifications in the form of B machines

- Reduce the time required to build process-control applications
- Reduce the effort required to build process-control applications
- Catch most of the Requirements bugs and enable the user to manipulate them before generating the formal specifications
- Enable the user to validate the requirements

We describe each of these five aspects in detail as follows.

A.2.1 Shorten the production-time of the applications

The required time to produce the requirements in final and correct form, modify them, and finally to translate them into formal specifications should be as minimal as possible. However, minimizing the time for production is not the only aspect because the economic cost, precision, reusability, and maintainability (of the produced requirements that can enable later access to the existing requirement in case there is a need for maintaining the system by adding new components or replacing existing ones) of the requirements should be considered as well. We suggest the reduction in time should be achieved through the following objectives:

- Having easy building blocks for the requirements; easy building blocks like standard rules with condition and action parts, invariants or goals can reduce the time required to build the application; however, a well-defined structure between these building blocks can effectively save more time.
- Increase requirements reusability, to enable the user to reuse existing requirements segments, when there is an opportunity, rather than creating them from scratch. This shortens the time from creating the whole application to the time required to create the other requirements segments (not found in the library) plus the time required to integrate the various requirements segments.

A.2.2 Reduce the production effort for applications

The effort and the expertise required to produce the requirements should be reduced as much as possible so that a normal systems engineer, who does not have detailed knowledge of formal specification structures or high-level logic and mathematics, can easily use the tool. Besides, we believe that the credibility of the tool can be increased when it saves the user effort when he needs to duplicate a number of goals or delete them, etc. In conclusion, we summarise the objectives that can reduce user effort as follows:

- Having easy building blocks for the requirements can reduce the effort required to build the application requirements. Also, having implied semantics for the structure over the building blocks will reduce the user's effort to arrange the building blocks.
- Gradually refined stages will help to split the building and debugging efforts into small successive levels rather than one level that is difficult to build and debug.
- Increase requirements reusability, to enable the user to reuse existing requirements segments when possible rather than creating them from scratch. This will save the user effort and enable him/her to focus on the application requirements rather than requiring more effort involving the construction of the application's components and high-level functions.

- Minimum knowledge of B and programming. Specific knowledge of B and formal methods is not convenient for the normal systems engineer to acquire. Thus; it would be easier to militate the required knowledge of formal methods and advanced mathematics and logic to basic understanding of the goal-model structure; this will result in a reduction of the effort required by a normal systems engineer to use the tool.

A.2.3 Need for formal specifications

The main required task of the tool is to aid in the gathering and analysis of the requirements and translating them into formal specifications. These required formal specifications will be refined by a software engineer from the programming point of view. A complier should be built within the GOPCSD tool used to translate the correct and checked requirements to B specification form.

A.2.4 Validate the user requirements

This objective is very important since a prototype is produced to provide the user with an understanding of how the application will behave. Although the alternatives to achieve this validation are many, we have to consider the most convenient alternatives that have a chance of being better understood by the systems engineer viewpoint. Also, because it is more likely that the systems engineer will find logical errors in his/her design, the tool should always correlate the validation result to corresponding requirements segments; thus the user will be able to correct them.

A.2.5 Catch Requirements Bugs

This analysis stage should contain exhaustive checking tests in order to eliminate as many bugs as possible and to not leave them for successive stages within the controller development lifecycle. To achieve this, we suggest the following objectives:

- Enable the user to validate and correct the requirements in relation to logical errors.
- Detect any inconsistency that exists in the system requirements and guide the user as to how to modify/correct the requirements to remove the inconsistency.
- Predict any obstruction that can happen at run-time and thus stop the application from operating in the properly designed manner.

From the above-discussed objectives, we can conclude that the GOPCSD tool should fulfil the following foundational objectives:

I. Using a goal-model as the requirements format, since it has simple building blocks (goals) and gradually refined stages (goal levels).
II. Provide a requirements segment library of the commonly used components and higher-level application functions.
III. Hide details of the B formal method.
IV. Automatically, translate the goal-models into B specification machines.
V. Enable the user to perform conflict analysis and guide the systems engineer in how to resolve the inconsistency.

VI. Enable the user to check the completeness and reachability of the goal-model and guide him/her in how to complete it.

VII. Enable the user to perform Obstacle analysis to detect the problems that can occur during run-time and guide him in how to eliminate or attenuate them if possible.

VIII. Provide an animation of the checked goal-models in order to validate them and to detect logical errors than will not be discovered in the other checking tests.

A.3 The GOPCSD tool entities

This section illustrates the data required to be stored in the library and the developed applications. As shown in figure A.5, the tool is composed of two main parts: one of them is the library that can be accessed by normal users for importing goal-models and components. It can be updated and renewed by a senior systems engineer. The second part is the environment for developing the requirements of the applications themselves, which is normally used by a systems engineer.

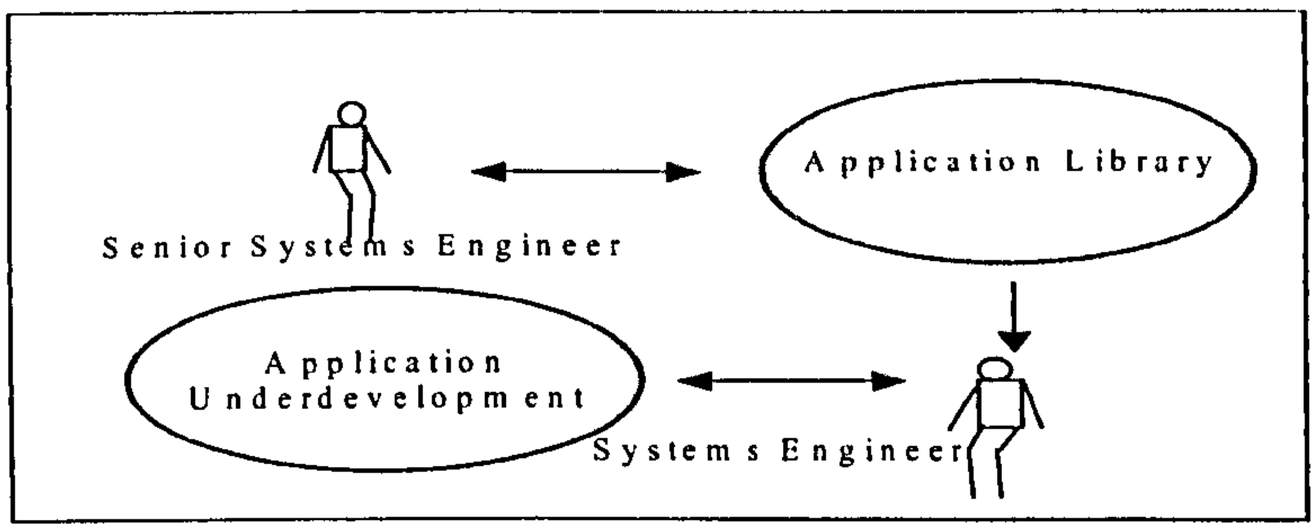


Figure A.5, the GOPCSD tool as the developed application and the library.

A.3.1 Entity–Relationship diagrams

Considering the number of goals, agents, variables and other tool objects that can be found within an application or within library applications, the tool can be better represented in terms of two separate databases: one database holding the library data and the other database for the application under development. In the following box, we list some statements about goal-models and process control systems that can help in modelling the two databases:

- Users specify goals; the goal can be refined into sub-goals or be a terminal goal.
- Agents are responsible for achieving terminal goals; an agent can be responsible for the accomplishment of more than one terminal goal.
- Goals contain variables that can be input, output or intermediate ones, while agents control output variables.
- The user can import components and templates from different libraries.
- A single library has a list of components and goal-model-templates, while component and goal-model templates belong to a single library.
- Applications have associated variables and agents.
- Library applications have components; the same component can be found in more than one application.

- Each component has a number of low-level goal-models that fully describe the operations of that component.
- The variables can have either normal types like (integer, real) or user-defined types that normally have values of the form (SWITCH_ON, SWITCH_OFF), i.e., an enumerated type.

A.3.1.1 Library system entities

From the above statements, the following entities can be identified for the library system: *{Library, Application, Component, High-level goal-model, Low-level goal-model, Goal, Agent, Variable, Variable Type, Variable Values}*. Such entities are related to each other through the relationships shown in table A.1 with an extra implied relationship between goals and variables that is many to many and non obligatory on both sides. This relationship is hardwired within each formal goal description that contains variables within it. It can be represent as a separate relationship; however, for simplicity it is not shown here. After determining the system entities and their relationships, the E-R diagram of the library system can be drawn as in figure A.6.

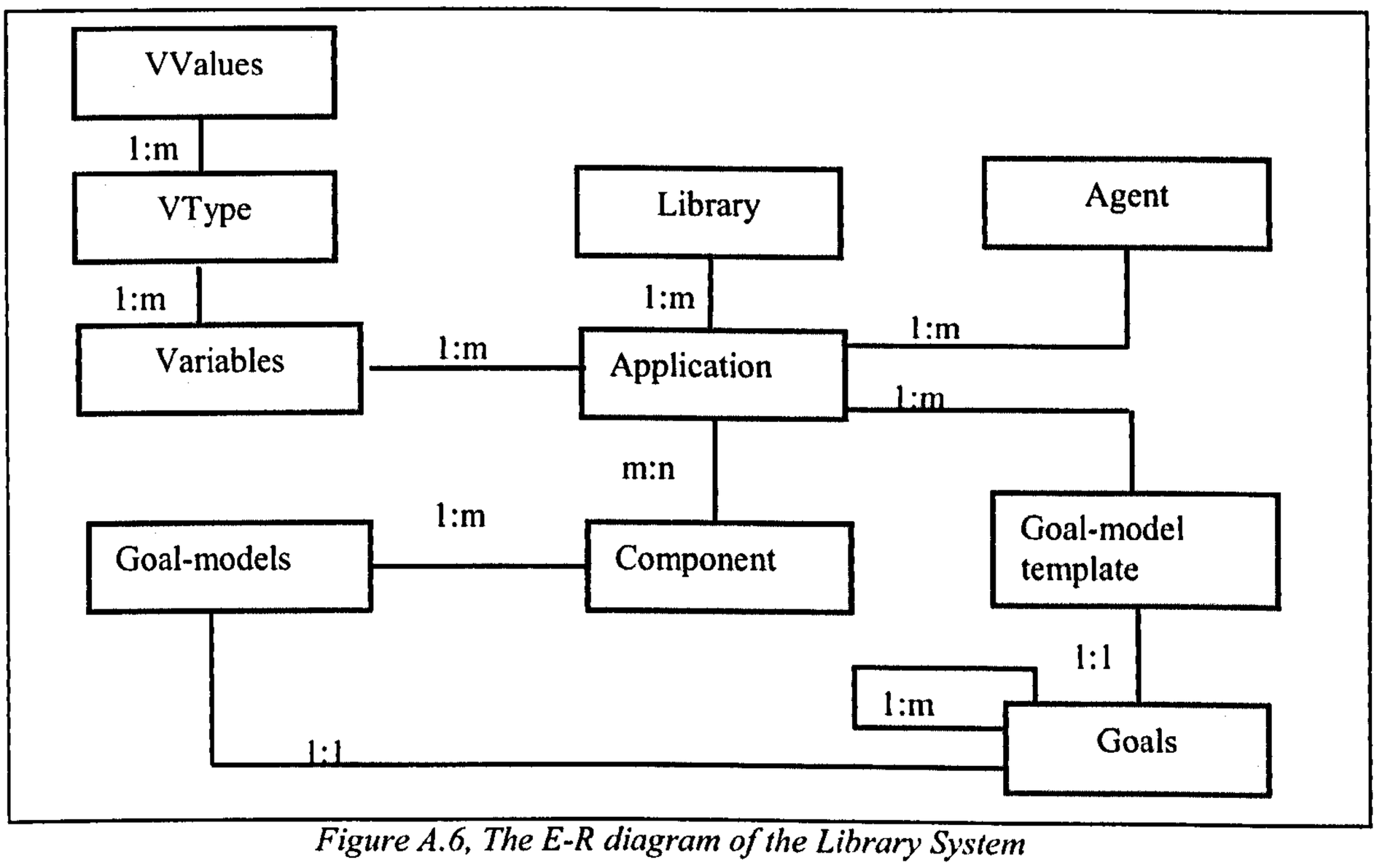


Figure A.6, The E-R diagram of the Library System

Table A.1, The relationships of the Library system

	Involved Entities of the relation	Relation Name	Aggregation	Obligation
L1	Library, Application	Contains	1:m	O:O
L2	Application, component	Contains	M:m	O:O
L3	Application, hl-goal-model	Contains	1:m	O:O
L4	Component, ll-goal-model	Contains	1:m	O:O
L5	Hl-goal-model, goal	Has_root	1:1	O:N
L6	ll-goal-model, goal	Has_root	1:1	O:N
L7	Agent, goal (only terminal)	Acomplishes	1:m	O:N
L8	Goal, Goal	Composed of	1:m	N:N
L9	Variable, Vtype	Has type of	M:1	N:O
L10	Vvalue, Vtype	Has value of	M:1	O:O

The details of the relationships of the library system are given as follows:

Relationship L1 "Contains" between Library and Application

- Each application belongs to exactly one library and each library must include at least one application. For example, relationship *Contains* relates an application called double-press production cell to a library called production cells.

Relationship L2 "Contains" between Application and Component

- Each component belongs to exactly one application and each application must include at least one component. For example, relationship *Contains* relates an application gas_burner to a component flame_detector.

Relationship L3 "Contains" between Application and high-level goal-model

- Each high-level goal-model belongs to exactly one application, but each application may include more than one high-level goal-model. For example, an application called simple production cell has a high-level goal-model called deposit-processed-metals.

Relationship L4 "Contains" between Component and low-level goal-model

- Each low-level goal-model, like open_valve or close_valve belongs to exactly one component, like valve, but the component can have more than one low-level goal-model.

Relationship "Has root" between low_level goal-model and goal

- This relation determines the root of each low-level goal-model. Each low-level goal-model has exactly one goal that is considered as its root goal. The goal appears only as a root of one goal-model. For example, a goal-model GM has one goal G1 as its root. The goal–model GM may contain other goals rather than G! which are refinement of goal G1; but, G1 cannot be a root for other goal-models and GM has no other goal as root.

Relationship L6 "Has root" between high-level goal-model and goal

- This relation determines the root of each high-level goal-model. Each high-level goal-model has exactly a single root goal; and each goal may be the root for exactly one high-level goal-model.

Relationship L7 "Accomplishes" between Agent and Goal

- The Accomplishes relationships represent the assignments of the agents to terminal goals; thus, it relates each terminal goal to a single agent. Each agent can be assigned to many terminal goals, but the same terminal goal cannot be assigned to more than one agent. For example, accomplishes can relate an agent gas motor to keeping the gas valve open goal.

Relationship L8 "Composed of" on Goal

- This relationship represented the refinement between the super and sub goals. From the super goal it relates a single goal and from the sub-goals it can be many goals. The relationship itself has attributes defining the refinement type; however since it is one to many, the relationship can be combined with the super goal relation; thus we will store the details of the relationship in the super goal row/ record.

Relationship L9 "Has Type of" between Variable and Vtype

- Each variable has exactly a single type, but many variables can have the same type. For example, the "Has type of" relationship can relate a variable like switch to a variable type ONOFF.

Relationship L10 "Has value of" between Vvalue and Vtype

- Each Variable type can have at least two values but each value should belong to a single type (if types are needed to be checked). For example, a type like ONOFF will have two rows in this

relationship, in one of them it relates the type to value ON and in the second it relates the type ONOFF to the value OFF.

A.3.1.2 Developed applications

Similar to the entities of the library system derived from the given statements, the developed application system has the following entities: {*Application, Component, High-level goal-model, Goal, Agent, Variable, Variable Type, Alternative solution, B-Machine, Obstacle*}. In table 2 we list the relationships between the different entities.

Table 2, The relationships of the developed applications

	Involved Entities of the relation	Relation Name	Relation Aggregation	Relation Obligation
A1	Application, hl-goal-model	Contains	1:m	O:O
A2	Application, alterantive	Has	1:m	O:N
A3	Alternative, Bmachine	Has	1:m	O:N
A4	Hl-goal-model, goal	Has_root	1:1	O:N
A5	Goal, goal	Composed of	1:m	N:N
A6	Agent, goal (only terminal)	Achieves	1:m	O:N
A7	Agent, Variable	Controls	1:m	O:N
A8	Variable, Vtype	Has type of	M:1	N:O
A9	Vvalue, Vtype	Has value of	M:1	O:O
A10	Obstacle, Goal	Obstructs	M:1	O:N

The relationships of the developed applications are explained in detail in the following:

Relationship A1 "Contains" between Application and high-level goal-model

- Each high-level goal-model belongs to exactly one application but each application should have at least one high-level goal-model. For example, the Contains relationship relates an application called gas-burner to a high-level goal-model called fulfil-user-requests.

Relationship "Has" between application and Alternative

- Each application may have more than one alternative solution but each alternative should belong to one application. For example, an application like gas-burner systems can have two alternative solutions: simultaneous alternative and sequence alternative.

Relationship A3 "Has" between Alternative and BMachine

- Each alternative solution has many B machines but each B machine belongs to exactly one alternative solution of one application. For example, if an application has different alternative solutions, each of them will have a list of B machines constituting the solution.

Relationship A4 "Has root" between high-level goal-model and goal

- This relation determines the root of each high-level goal-model. Each high-level goal-model has exactly a single root goal; and each goal may be the root for exactly one high-level goal-model. This relationship is similar to L6 of the library system.

Relationship A5 "Composed of" on Goal

- This relationship represents the refinement between the super and sub goals. It relates one single super-goal to many sub-goals. But, each sub-goal is related to exactly one super-goal. This relationship is similar to relationship L8 of the library sub-system.

Relationship A6 "Accomplishes" between Agent and Goal

The Accomplishes relationships represent the assignments of the agents to terminal goals; thus, it relates each terminal goal to a single agent. This relationship is similar to relationship L7 of the library sub-system.

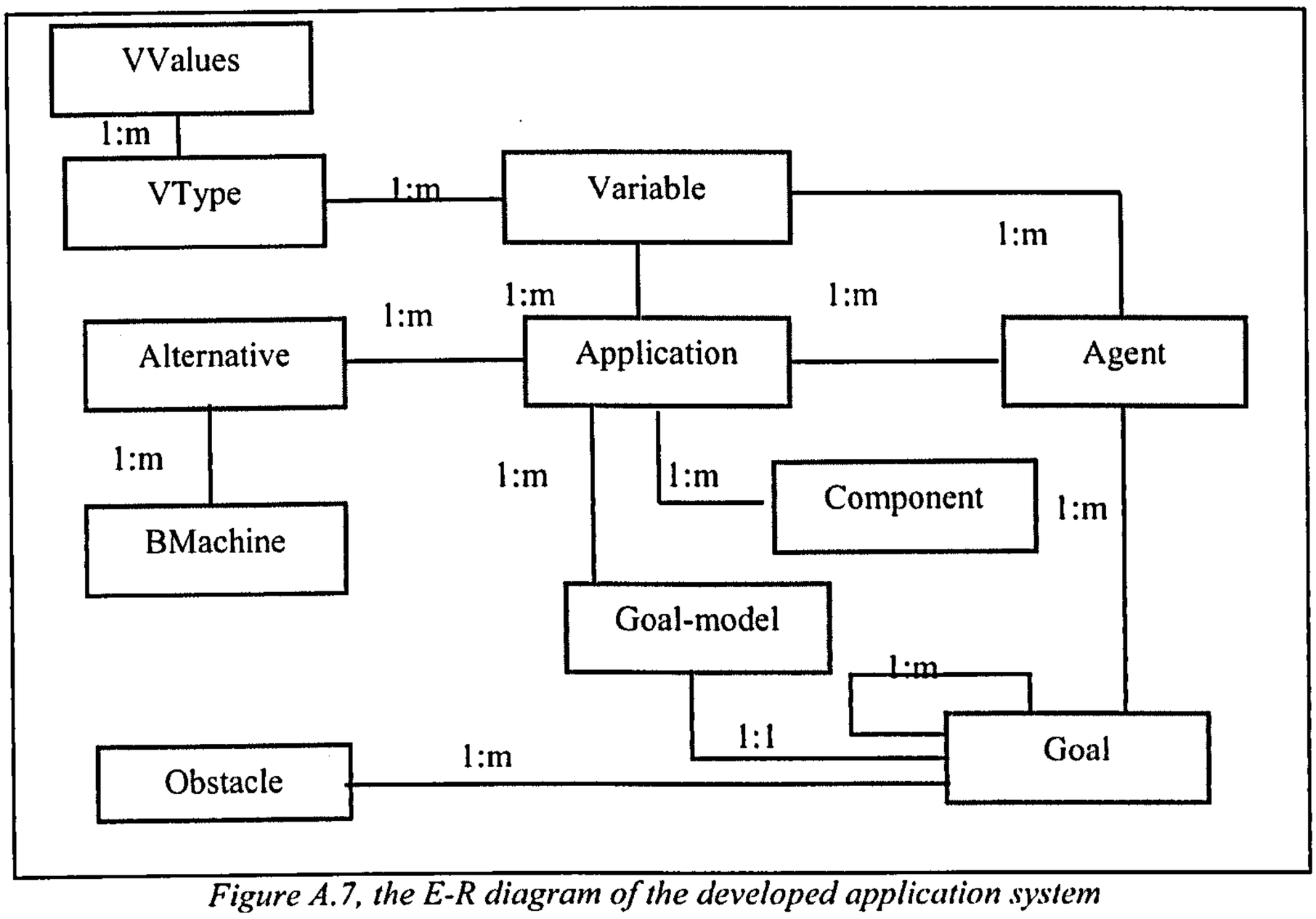


Figure A.7, the E-R diagram of the developed application system

Relationship A7 "Controls" between Agent and Variable

- This relation determines which agent controls which variable. For uniqueness of control, only a given variable will be controlled by exactly one agent, but an agent can control more than one output variable.

Relationship A8 "Has Type of" between Variable and Vtype

- Each variable has exactly a single type, but many variables can have the same type. This relationship is similar to L9 of the library system.

Relationship A9 "Has value of" between Vvalue and Vtype

- Each Variable type can have at least two values but each value should belong to a single type (if types are needed to be checked). This relationship is similar to L10 of the library system.

Relationship A10 "Obstructs" between Obstacle and Goal

- Each Gaol can be obstructed by more than one obstacle but we will restrict the design to duplicate the obstacle if it obstructs different goals.

Providing the entities and their relationships, the E-R diagram of the Developed application can be seen in figure A.7.

A.3.2 Translating the E-R diagram into relations

The ERD diagrams of the Library and developed application system can be translated into BCNF relations [Date, 86] in order to represent the main data required to be stored by the tool.

A.3.2.1 Library sub-system relations

The ERD diagram of figure A.6 is translated into database relations of BCNF. The linkages between the relations are shown in figure A.8 as the lines between the common fields between the relations. The following are the relations of the library system:

A.3.2.1.1 Library relation

This relation stores the different libraries of process control families.

Library			
	Field name	Comments	Type
1	L_id	Library Identifier	Number
2	L_name	Library name	Text
3	L_desc	Library description	Text

A.3.2.1.2 Application relation

This relation stores the different applications of each library.

Application			
	Field name	Comments	Type
1	App_id	Application Identifier	Number
2	l_id	Owner Library id	Number
3	App_name	Application name	Text
4	Appl_desc	Application description	Text

A.3.2.1.3 Component relation

The different components of the process control systems are stored within this relation. The different applications can be composed of similar components.

Component			
	Field name	Comments	Type
1	C_id	Component Identifier	Number
2	C_name	Component Name	Text
3	C_desc	Component Description	Text

A.3.2.1.4 App_Com relation

This relation represents the relationship of type many to many between the applications and the components.

App_Com			
	Field name	Comments	Type
1	App_com_id	App_com Identifier (many-many relationship)	Number
2	App_id	Owner Application Identifier	Number
3	C_id	Owner Component Identifier	Number

A.3.2.1.5 High-level goal-model relation

This relation stores the goal-model templates for the different applications.

Hl-gm			
	Field name	Comments	Type
1	Hl_gm_id	High-level goal-model Identifier	Number
2	Hl_gm_name	High-level goal-model name	Text
3	Hl_gm_desc	High-level goal-model description	Text
4	Root_id	Root goal Identifier	Number
5	App_id	Owner Application Identifier	Number

A.3.2.1.6 Low-level goal-model relation

The different operations specifying a stored component in the library can be represented by a list of goal-models; each operation will be represent by a low-level goal-model because it will appear in the whole application at the lower-level, closer to the agent assignments. The component may have more than one goal-model.

Ll-gm			
	Field name	Comments	Type
1	Ll_gm_id	Low-level goal-model Identifier	Number
2	Ll_gm_name	Low-level goal-model name	Text
3	Hl_gm_desc	High-level goal-model description	Text
4	Root_id	Root goal Identifier	Number
5	C_id	Owner Component Identifier	Number

A.3.2.1.7 Goal relation

The goal information is stored in this relation in addition to the sub-super recursive relationship within the goals.

Goal			
	Field name	Comments	Type
1	G_id	Goal Identifier	Number
2	G_name	Goal name	Text
3	G_desc	Goal description	Text
4	G_fdesc	Goal formal description	Text
5	Pg_id	Parent goal identifier	Number
6	G_c_desc	Goal condition description	Text
7	G_c_fdesc	Goal condition formal description	Text
8	G_terminal	Is the Goal Terminal or not	Boolean
9	G_refinement	Goal refinement type (sequence,..)	Number
10	G_order	A number used for ordering the sub-goals	Number
11	G_tempo	Temporal (0= maintain /1= achieve /2= avoid /3= cease)	Number

A.3.2.1.8 Agent relation

The agents of each application are stored in this relation. Links from terminal goals can be used in the case of imported components or self-links (app_id) can be used in the case of imported library agents.

Agent			
	Field name	Comments	Type
1	A_id	Agent Identifier	Number
2	App_id	Owner Application Identifier	Number
3	A_desc	Agent Description	Text
4	A_name	Agent name	Text
5	A_type	Agent type (1= device/ 2=software/ 3=human)	Number

A.3.2.1.9 Variable relation

This relation stores the different variables with their types for the library applications. When a tool user imports a variable from the library it may be duplicated by having different instances of the class variable in the library.

Variable			
	Field name	Comments	Type
1	V_id	Variable Identifier	Number
2	App_id	Owner Application Identifier	Number

3	T_id	Type Identifier (-1=integer / –2=real)	Number
4	V_name	Variable name	Text
5	V_desc	Variable Description	Text

A.3.2.1.10 Variable type relation

In this relation, the variables' types are stored; when the user creates a variable with enumerated type, like bi-state and tri-state, the variable type identifier T_id will be stored within the variable details.

Vtype			
	Field name	Comments	Type
1	T_id	Type Identifier	Number
2	T_name	Type name	Text
3	T_desc	Type description	Text

A.3.2.1.11 Type value relation

This relation mainly contains the enumerated type values like ON, OFF for different variables.

Vvalue			
	Field name	Comments	Type
1	Vv_id	Variable Value Identifier	Number
2	T_id	Type Identifier	Number
3	VV_desc	Variable value	Text

A.3.2.2 Developed Application relations

The E-R diagram of figure A.6 is translated into database relations of Boyce/Codd Normal Form(BCNF). The linkages between the relations are shown in figure A.8 as the lines between the common fields between the relations.

A.3.2.2.1 Application relation

This relation holds the general information about each application the tool user develops.

Application			
	Field name	Comments	Type
1	App_id	Application Identifier	Number
2	App_name	Application name	Text
3	Appl_desc	Application description	Text

A.3.2.2.2 High-level goal-model relation

These are the incomplete goal models created by the tool user or imported from component importing.

Hl-gm			
	Field name	Comments	Type
1	Hl_gm_id	High-level goal-model Identifier	Number
2	Hl_gm_name	High-level goal-model name	Text
3	Hl_gm_desc	High-level goal-model description	Text
4	Root_id	Root goal Identifier	Number
5	App_id	Owner component Identifier	Number

A.3.2.2.3 Goal relation

This relation has the information about the separate goals and the links between them that represents the tree structure.

Goal			
	Field name	Comments	Type
1	G_id	Goal Identifier	Number
2	G_name	Goal name	Text
3	G_desc	Goal description	Text
4	G_fdesc	Goal formal description	Text
5	Pg_id	Parent goal identifier	Number
6	G_c_desc	Goal condition description	Text
7	G_c_fdesc	Goal condition formal description	Text
8	G_terminal	Is the Goal Terminal or not	Boolean
9	G_refinement	Goal refinement type	Number
10	G_order	A number used for ordering the sub-goals	Number
11	G_tempo	Temporal (0=maintain/1=achieve/2=avoid /3=cease)	Number

A.3.2.2.4 Agent relation

This is the agent relation. The relation has information about the type, description and name of each agent.

Agent			
	Field name	Comments	Type
1	A_id	Agent Identifier	Number
2	App_id	Owner Application Identifier	Number
3	A_desc	Agent Description	Text
4	A_name	Agent name	Text
5	A_type	Agent type (1= device/ 2=software/ 3=human)	Number

A.3.2.2.5 Variable relation

The application variables will be stored in this relation, both the variables imported by the tool user from the library and the newly created ones.

Variable			
	Field name	Comments	Type
1	V_id	Variable Identifier	Number
2	App_id	Owner Application Identifier	Number
3	T_id	Type Identifier (-1=integer / –2=real)	Number
4	V_name	Variable name	Text
5	V_desc	Variable Description	Text
6	V_type	Variable type (0=input/ 1=output? 2=integer)	Number

A.3.2.2.6 Variable type relation

The user-defined variables will be stored within this relation as well as imported variable types of non-standard types (neither integer nor real).

Vtype			
	Field name	Comments	Type
1	T_id	Type Identifier	Number
2	T_name	Type name	Text
3	T_desc	Type description	Text

A.3.2.2.7 Type value relation

The different enumarated values of the user defined types, each value of specific type, will be stored in a database row. For example, when the user creates a tri-state variable type that has three values, three rows will be added to this relation.

Vvalue			
	Field name	Comments	Type
1	Vv_id	Variable Value Identifier	Number
2	T_id	Type Identifier	Number
3	VV_desc	Variable value	Text

A.3.2.2.8 Alternative relation

As there is an alternative refinement pattern there will more than one final goal-model implying possiblly more than one solution. Each alternative record in the database denotes one solution of some application.

Alternative			
	Field name	Comments	Type
1	Alt_id	Alternative Identifier	Number
2	Alt_name	Alternative Name	Text
3	Alt_desc	Alternative description	Text
4	App_id	Owner application Identifier	Number

A.3.2.2.9 B Machine relation

This relation stored the file names of the B machine Specifications. The text of the specification is not stored within the database, but the name of the text file that contains the B specifcation machine and is stored in the application directory.

Bmachine			
	Field name	Comments	Type
1	Bm_id	B machine Identifier	Number
2	BM_name	B machine name	Text
3	BM_desc	B Machine Description	Text
4	BM_fn	B Machine file name	Text

A.3.2.2.10 Obstacle relation

This relation stores obstacles that may obstruct goals, mostly terminal goals, from achieving the desired functions. A reference to the obstructed goal (G_id) will be stored with each obstacle.

Obstacle			
	Field name	Comments	Type
1	O_id	Obstacle Identifier	Number
2	O_name	Obstacle name	Text
3	O_desc	Obstacle description	Text
4	G_id	Obstructed Goal Identifier	Number

A.4 A tutorial for the GOPCSD tool

This section can be regarded as the main guideline to construct structured requirements for software applications, and in particular, to formally specify Process Control Systems.

A.4.1 The requirement elements

Before we can describe the tool's operation, it would be helpful to provide some background of the elements we use to store the user's requirements. In the GOPCSD tool, we store the application's requirements using components to describe the structure of the application, variables to describe the application state, agents to describe the active entities to change the application state, and goal-models to specify how to control the application.

A.4.1.1 Components

Components represent the physical parts of the applications: examples include valves, robots, and deposit belts. Their specifications are stored in the Library; in the GOPCSD tool, we recommend the user to import, where possible, the standard components from the provided library rather than creating them from scratch. Having imported the components, all of their related agents, variables and goal-models will be added to the application space and, consequently, will promptly appear in the corresponding agents, variables and goal-models lists. The GOPCSD environment tool does not support the creation of the components for consistency reasons. However, the user can still create the library components and specify their details in the GOPCSD Library Manager program. The details of importing the components and mapping their variables and agents are explained in section A.1.

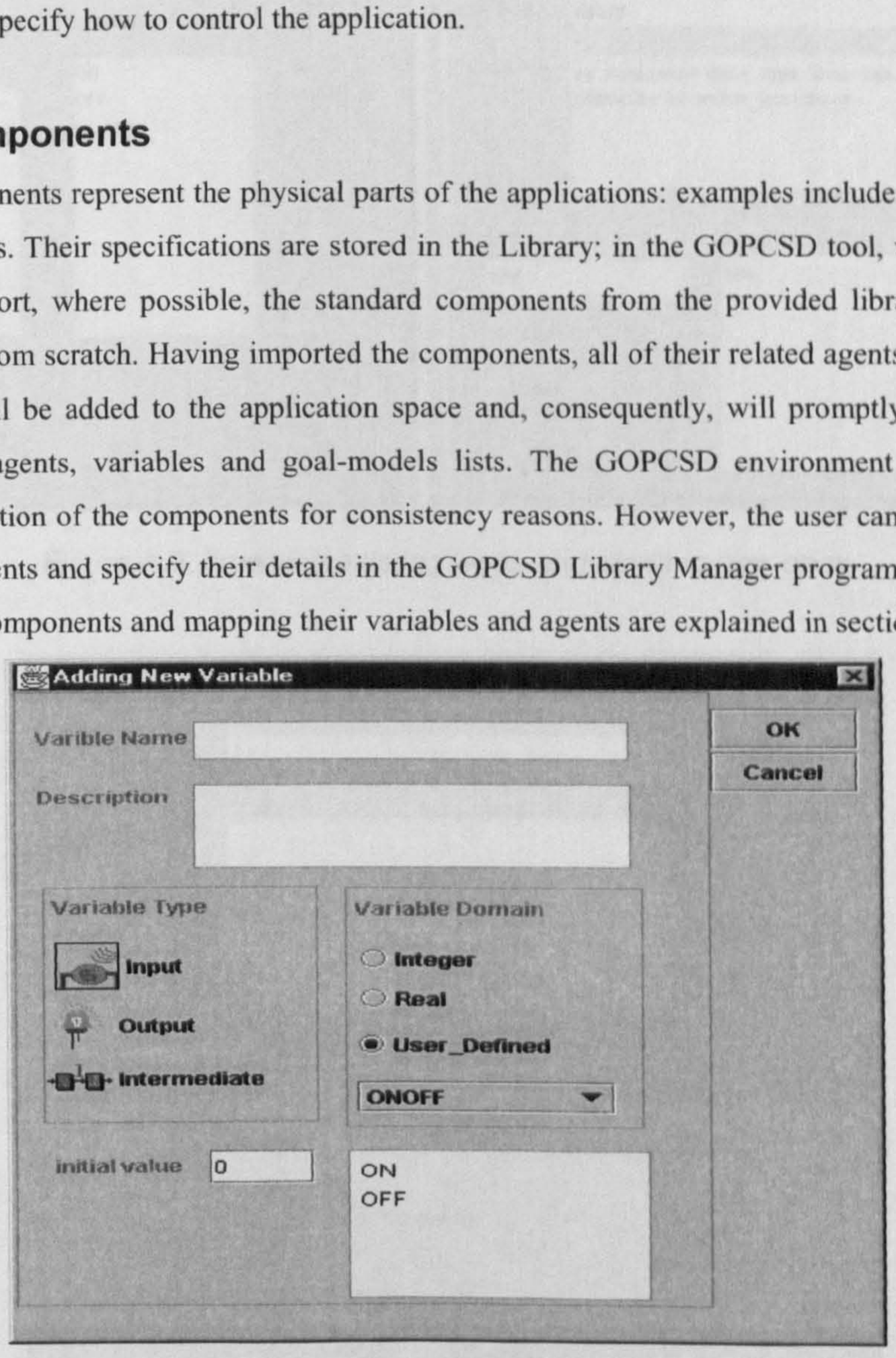


Figure A.8, the dialogue box to create new variable

A.4.1.2 Variables

Variables are considered as an essential aspect in formalising the user requirements. In the GOPCSD tool, the application's global state is normally described by a set of variables. Each of these

variables has one of three types: input, output and intermediate. In the GOPCSD, the variables are associated with the high-level goal-model templates or the components, which the user imports from the library; however, the tool's user can also create, edit, and delete variables from the application space. Figure A.8 shows the Dialogue Box that the user uses to edit variables.

The types that can be assigned to the variables can be integer, real or enumerated range values like ONOFF. Figure A.9 shows the dialogue box the tool employs to enable the user to add new data types.

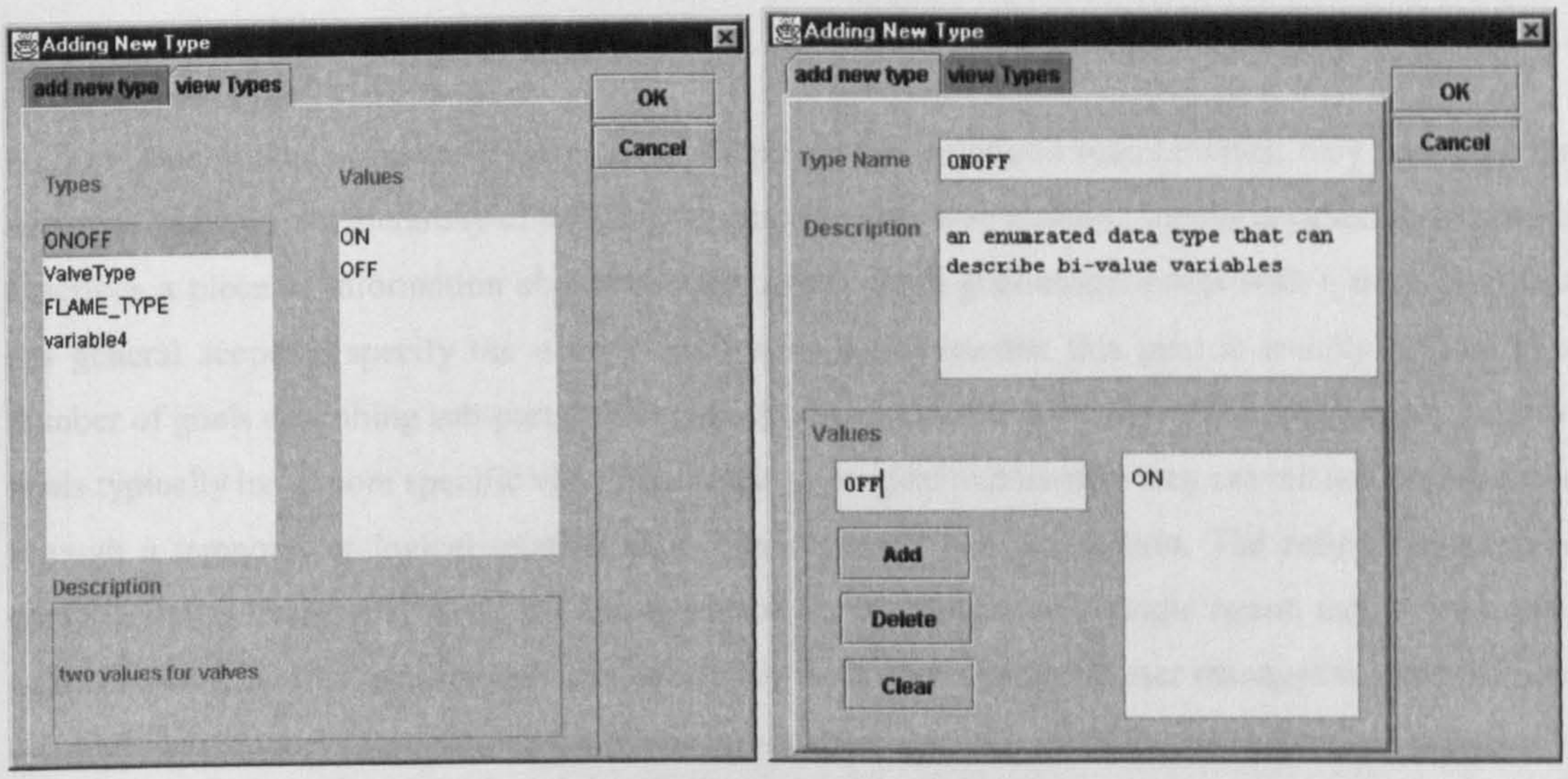


Figure A.9, browsing existing types and creating new ones

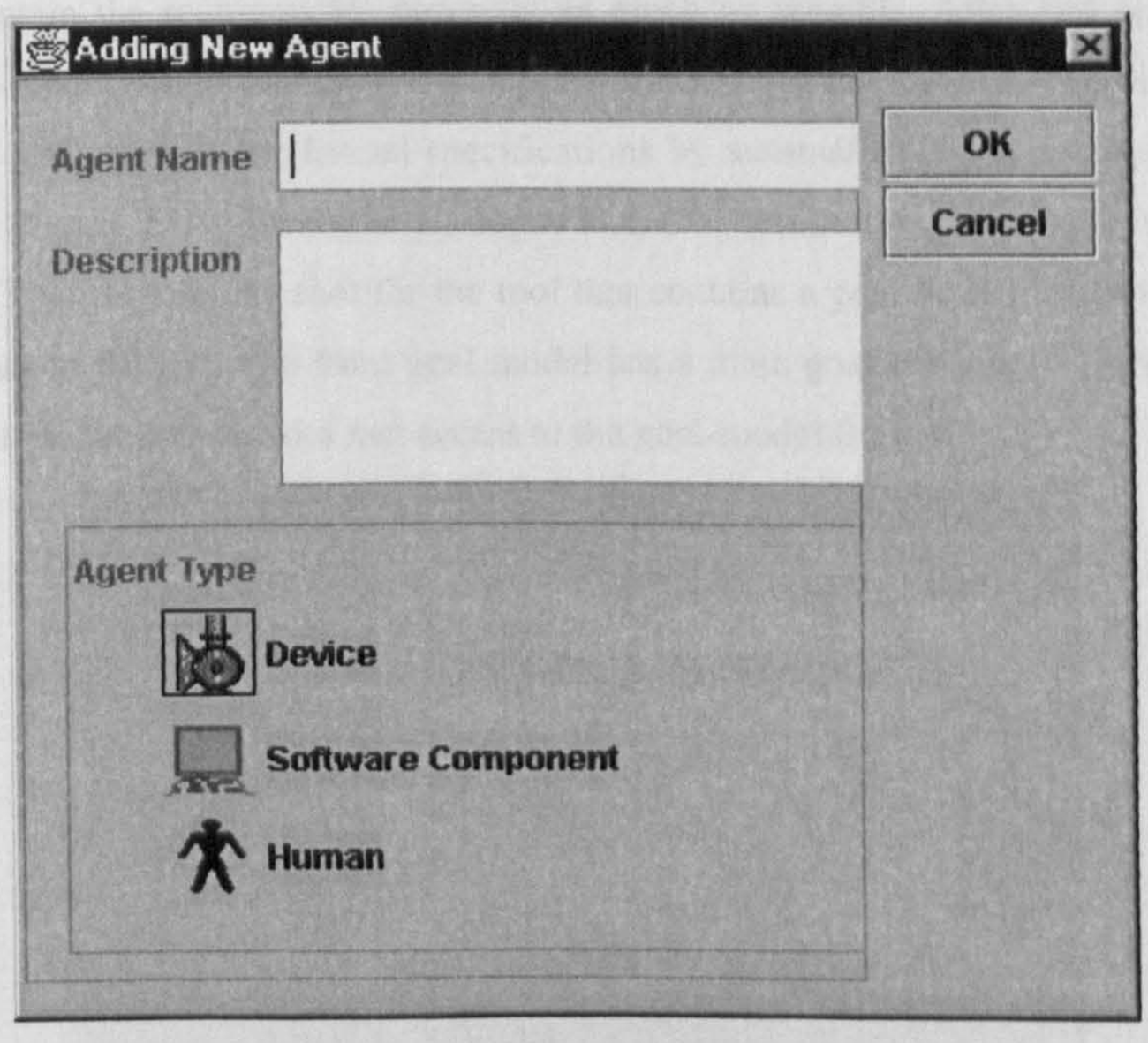


Figure A.10, the dialogue box to create new agent

A.4.1.3 Agents

Agents are the objects that control the application and its local environment. Some of the agents can be part of the application to be built, like software interface programs for hardware parts, or,

alternatively, they can be existing programs or hardware devices that will be responsible for accomplishing pre-defined tasks (goals) to fulfil the overall application operation. Agents can have one of the following three types: devices, software, and human. The main source of agents is the user importing components from the library. But, if it is required to declare agents apart from those associated with the components, the user can create, edit, and delete them from within the GOPCSD tool environment. Figure A.10 shows the New Agent Dialogue Box where the user can add agents to the application.

A.4.1.4 Goal-models

Goal-models constitute the main segments of the structured requirements; they represent the user requirements in a hierarchy of building blocks; each of these building blocks is called a goal and it describes a piece of information about the application. Each goal-model starts with a main goal that has general scope to specify the overall application requirements; this goal is usually refined to a number of goals describing sub-parts, different aspects, or operation-modes of the application; the sub-goals typically have more specific views than their super goal and usually they are related to each other through a temporal or logical relationships, like sequence and disjunction. The refinement process continues until every goal is simple enough to be accomplished by a single agent, called a terminal goal. In GOPCSD tool, goal-models can be used as workspaces until the user manages to construct one complete goal-model that specifies the whole application and each of its terminal goals are assigned to an agent.

The goal-model can be checked as soon as it is created in order to provide the user with feedback to maintain the requirements correctly as much as possible. After this repeated feedback process, the goal-model can be animated to catch the logical bugs that could not be discovered earlier. Finally, the user can generate the formal specifications by automatically translating the goal-models into B machines. Figure A.11 shows the Dialogue Box that can be used to create a new goal-model, while figure A.12 shows a screen shot for the tool that contains a goal-model list to the right and two goal-model frames to the left. The front goal model has a main goal G1 refined to two sub-goals G2 and G3. The goal-model list provides fast access to the goal-model frames.

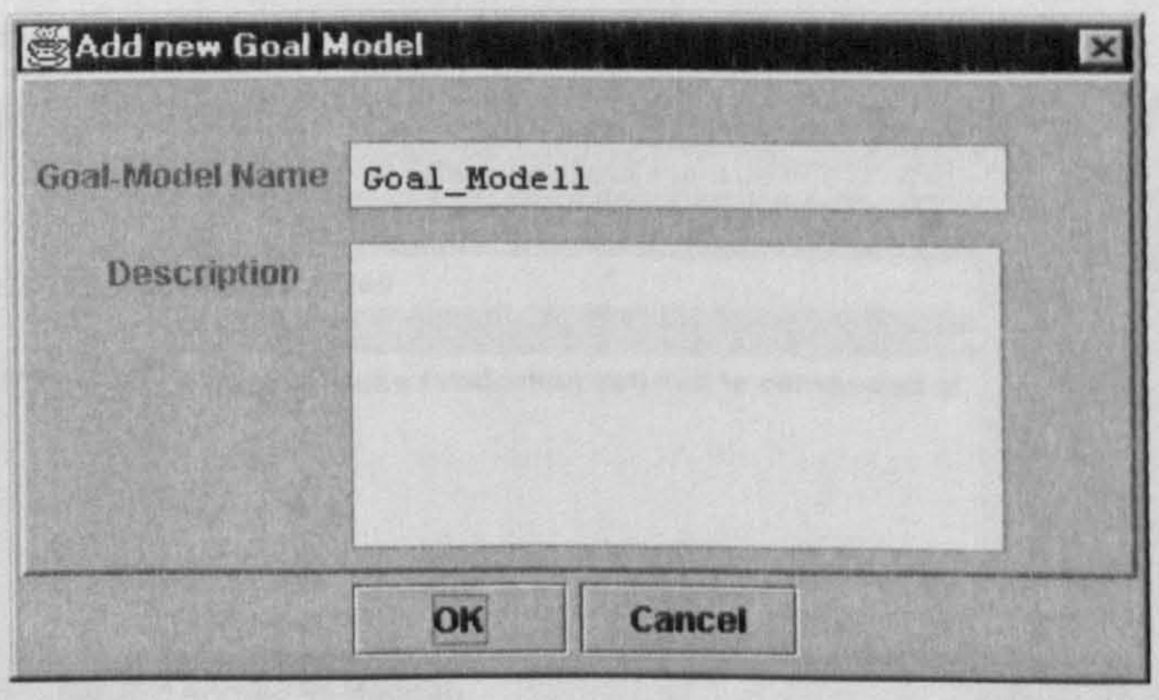


Figure A.11, creating new goal-model as workspace.

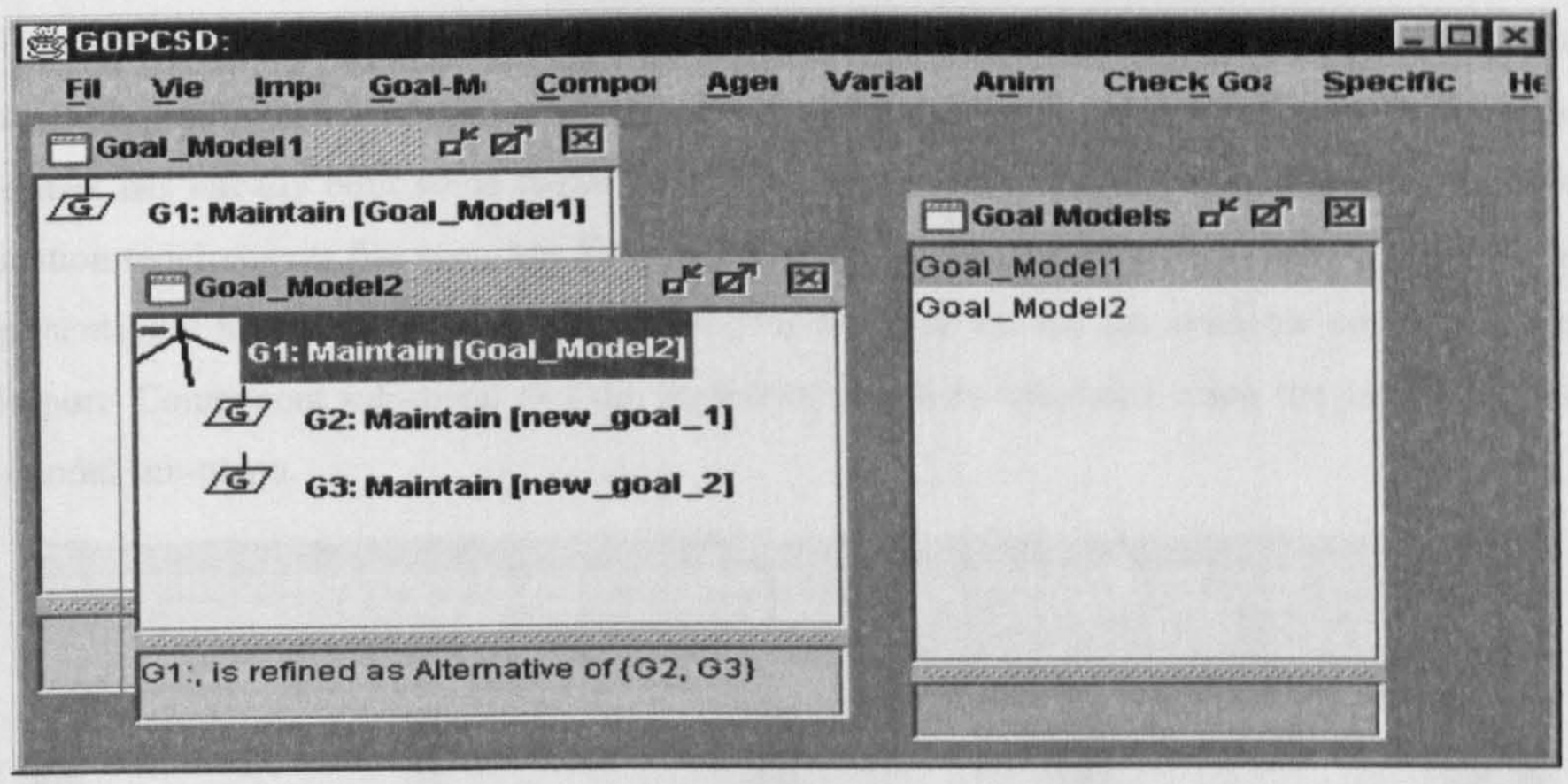


Figure A.12, the GOPCSD desktop contains goal-model list to the right and two goal-model frames to the left.

After providing this background, we can now proceed to describe how to use the tool to build process control applications. There are three phases for any application to be constructed by the tool: construct the goal-model, check the goal-model and modify it, and finally, automatically translate the goal-model to B machines. In sections 4.2, 4.3 and 4.4 we describe how the tool can be used in these three phases, respectively.

A.4.2 Phase I, Construct the goal-model

In the first phase, the user constructs the application; the tool provides guidance throughout this phase. In the following sub-sections, we provide a brief description of how to use the tool to build a complete goal-model. Although the sub-sections may be considered as a sequence of steps, the user can repeat some of them.

A.4.2.1 Starting New Application

When the user creates the application for the first time, he should select the file menu and then new in order to document the application to be created. Figure A.13 shows the New Application Information dialogue.

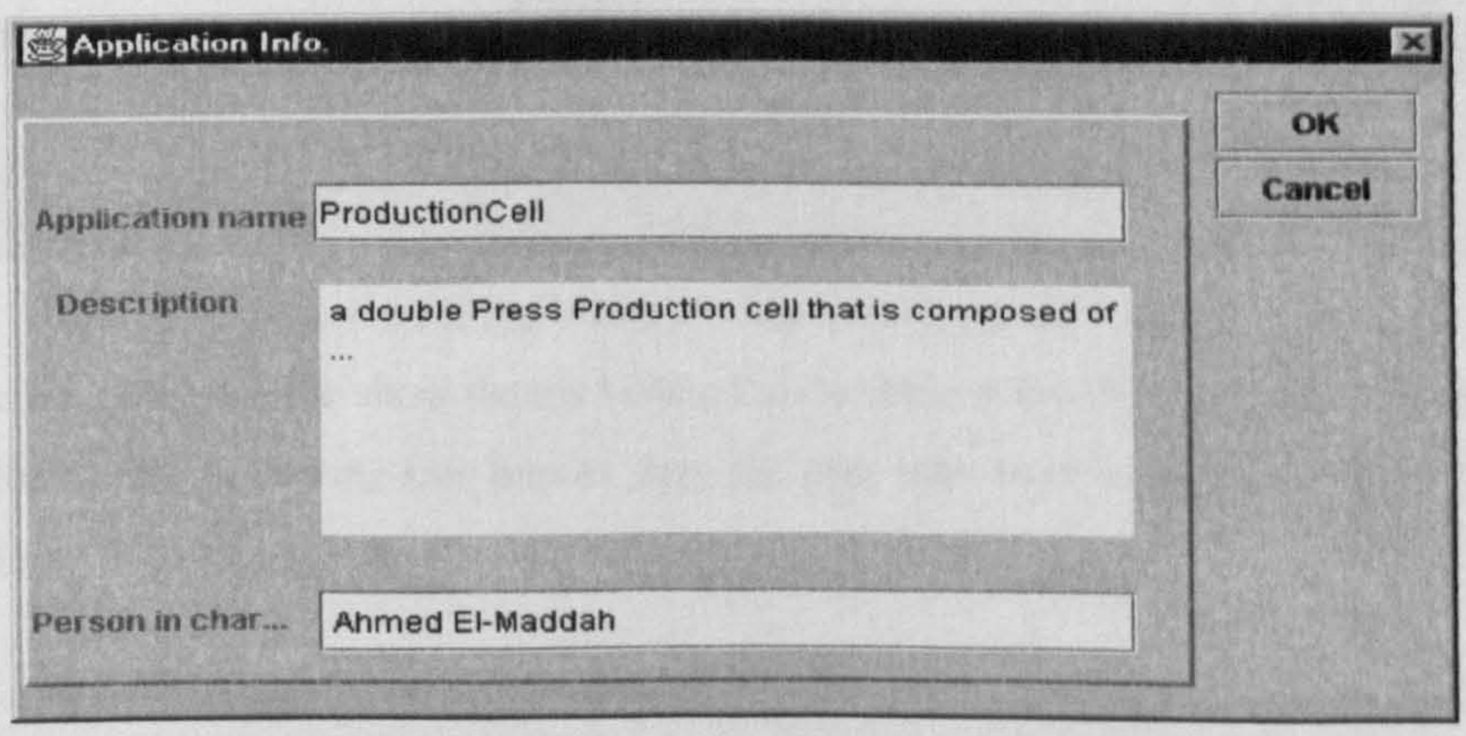


Figure A.13, creating new application Dialogue box

After the user documents the new application, the GOPCSD desktop contains four lists for accessing the application: Components, Agents, Variables and Goal-models as shown in figure A.14. If the user has already built some requirements and saved them, he/she can open the corresponding application requirements file using the File/Open sub-menu. The next step is to import the application components and high-level function templates. The user can browse the available components using the Import/ Component sub-menu and the high-level function templates using the Import/high-level goal-model sub-menu.

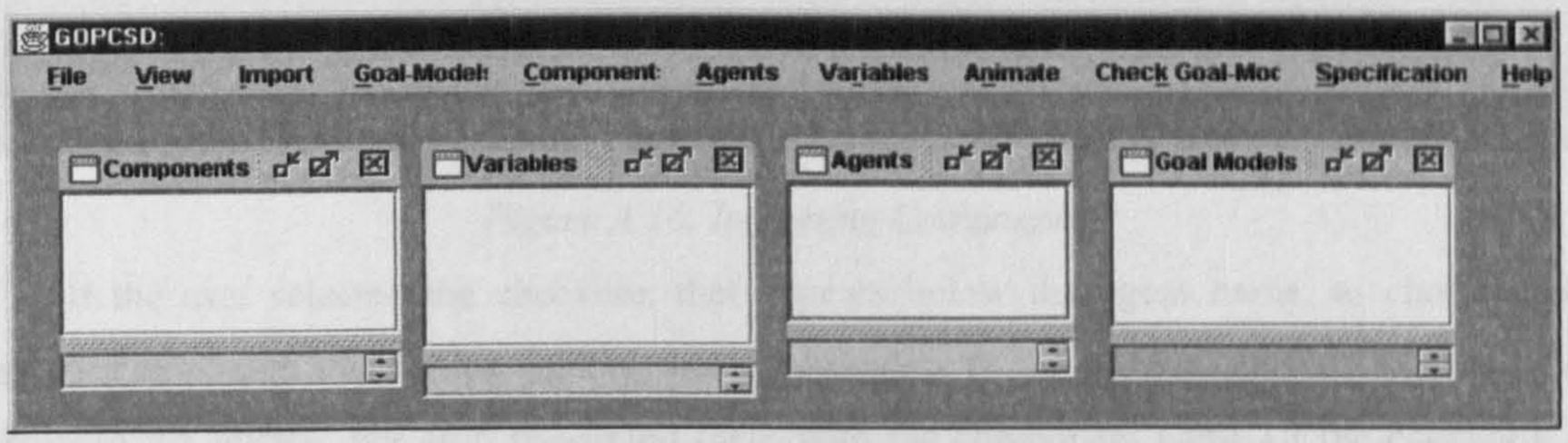


Figure A.14, the GOPCSD desktop with empty lists

A.4.2.2 Importing a Component from the Library

A.4.2.2.1 Selecting the component

After the user identifies the application components and locates them within the library, he/she selected the corresponding component by using the dialogue box shown in figure A.15.

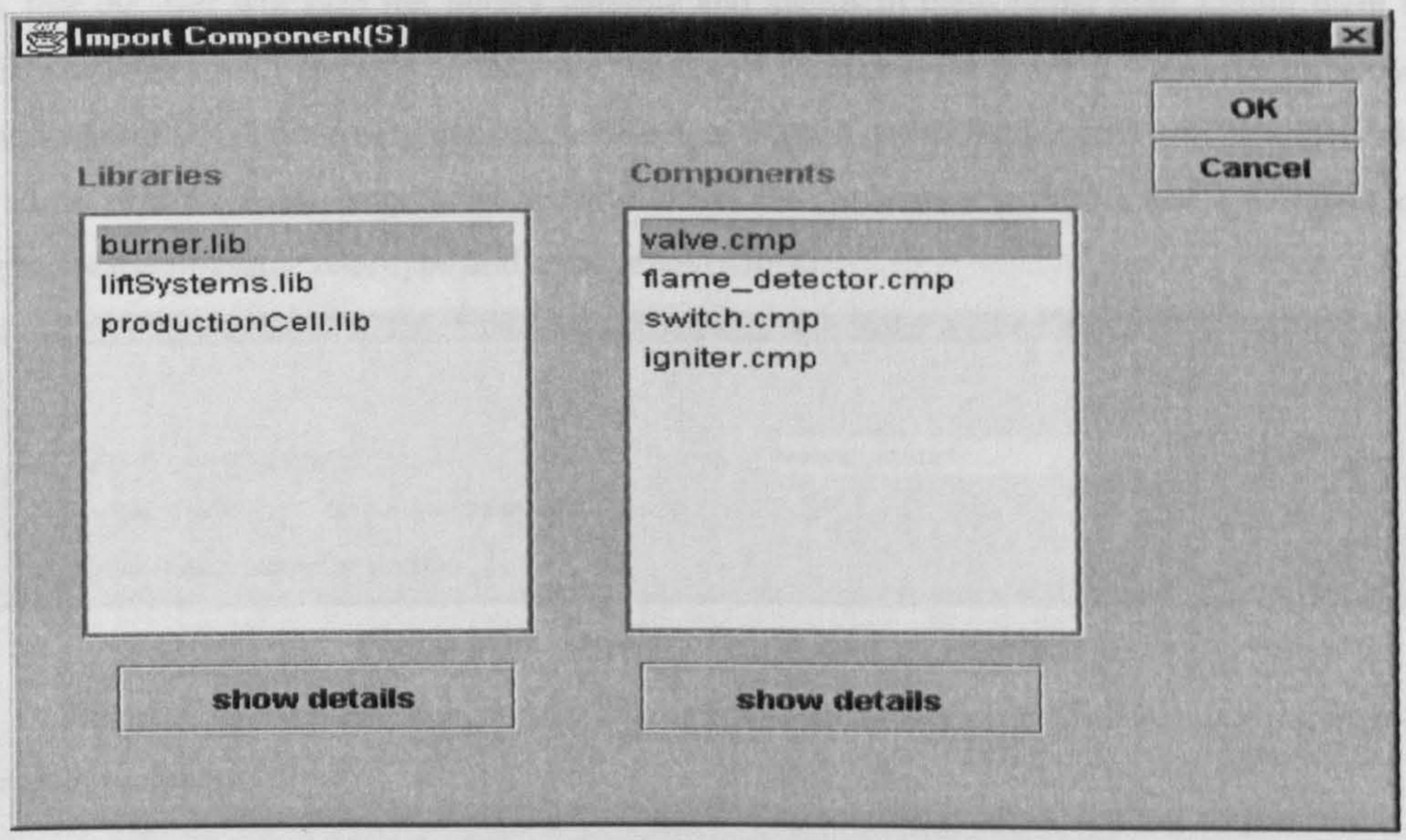


Figure A.15, importing a component from the library

The user may use the show details button located below the libraries and components lists to get extra information. Following this import step, the user may have to map or rename the variables and the agents as follows.

A.4.2.2.2 Renaming the Component

The user is allowed to rename the imported component to have a more comprehensive style of naming the components with functional names rather than enumerating them; the user can change the

component name from the standard name used in the library, especially if the application contains multiple instances of the same component. Following the component import dialogue in figure A.15, the dialogue in figure A.16 appears to allow the user to rename the component.

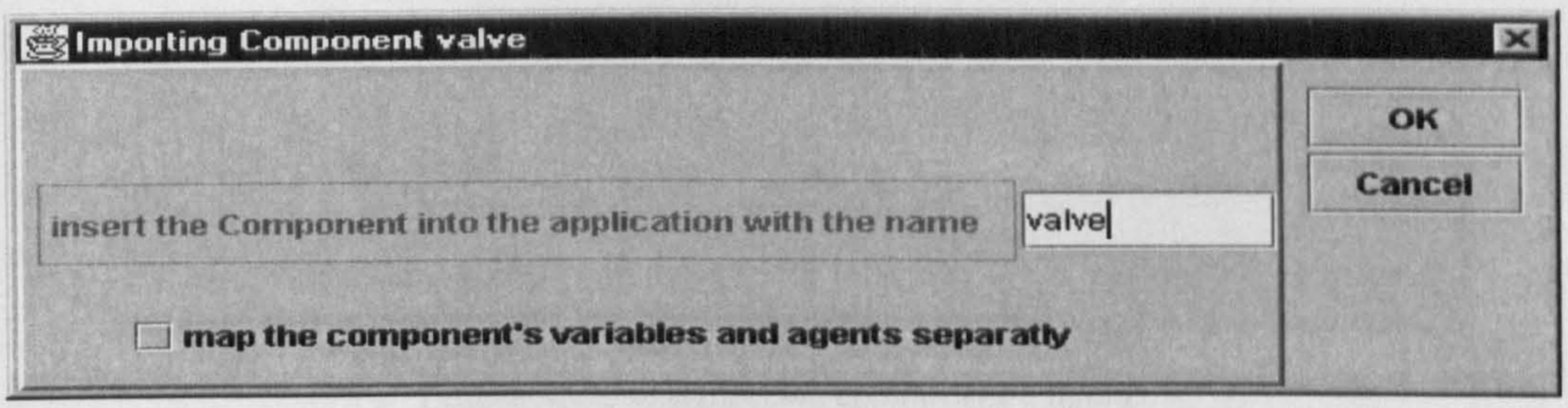


Figure A.16, Importing Component

If the user selected the checkbox that appears below the agent name, to choose between adding the component's variables, agents, and goal-models to the application with the same names they have in the library, but after pre-fixing them with the component name (if the checkbox is not selected), or to map each variable and agent in turn.

A.4.2.2.3 Mapping Component's Variables

Sometimes, after selecting the component, the user wants to rename each variable and agent in turn. This preference can arise in the case where some of the variables or the agents names already exists, that the user will map the library variable and agents to them rather than adding them to the application. Figure A.17 shows a dialogue box to map a component's variable. There are three choices: the variable will be added to the application with a new typed name, the variable will be added with the same name, or the user can map the variable to one of the application variables that is compatible with the library variable (same data-type and same input/output).

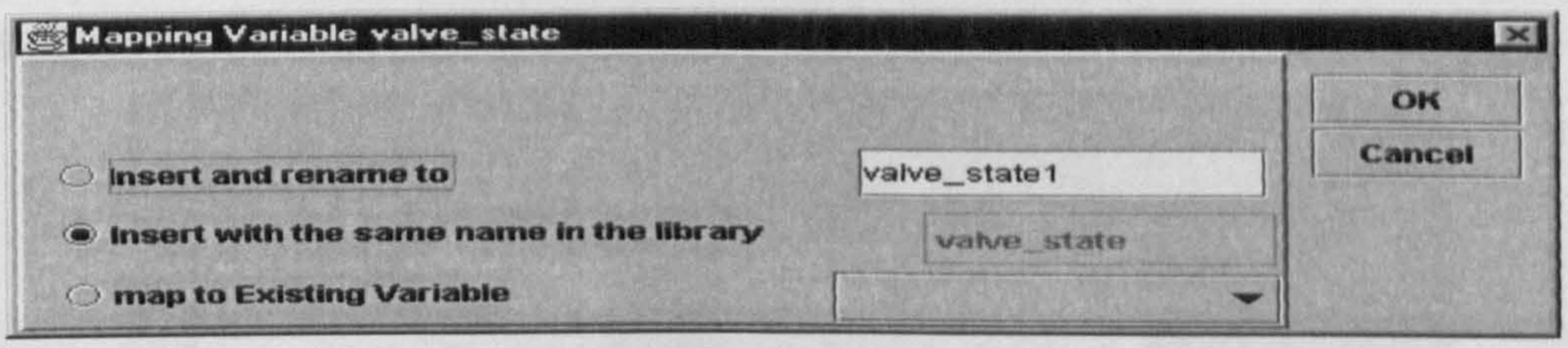


Figure A.17, Mapping Component's variables

This dialogue appears after importing a component and very rarely after importing high-level goal-model templates.

A.4.2.2.4 Mapping the Component's Agents

After the user maps the components' variables, he now needs to map the agents that controls the variables and are associated with the goal-models. Figure A.18 shows map agents dialogue. It is similar to the dialogue in figure A.17. This dialogue box appears after importing a component and, very rarely, after importing high-level goal-model templates. Figure A.19 shows the desktop of the tool after importing a valve component. There is one variable *valve_state* in the variable list, one agent, motor, in the agent list, one component valve in the components list and two goal-models, *closeValve*

and *openValve,* in the goal model list; these two goal-models appear in separate internal frames as *G1:Achieve[openValve]* and *G1:Achieve[closeValve].* The user can import the rest of the component in the way as described in this section.

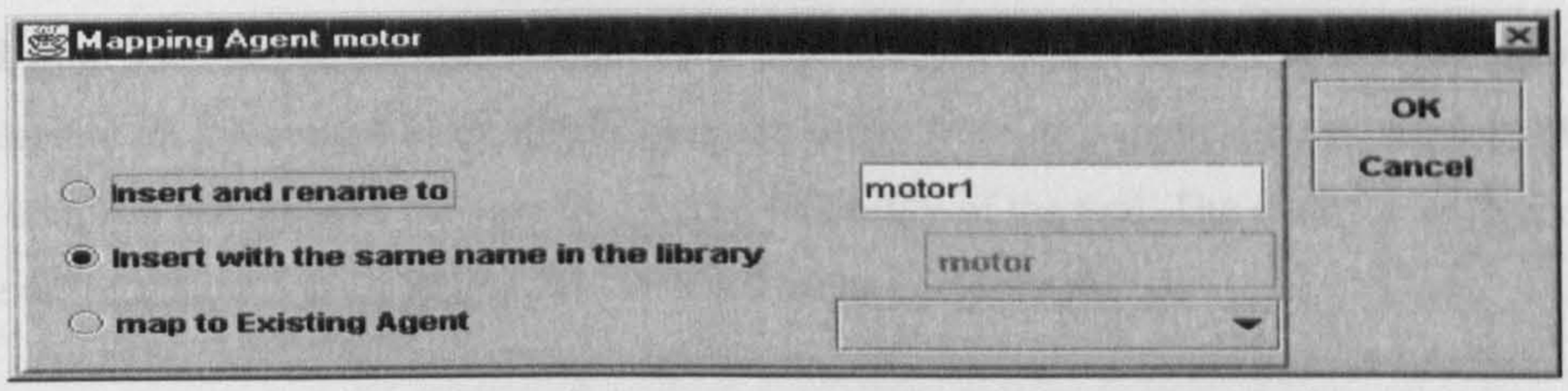


Figure A.18, mapping the component's agents

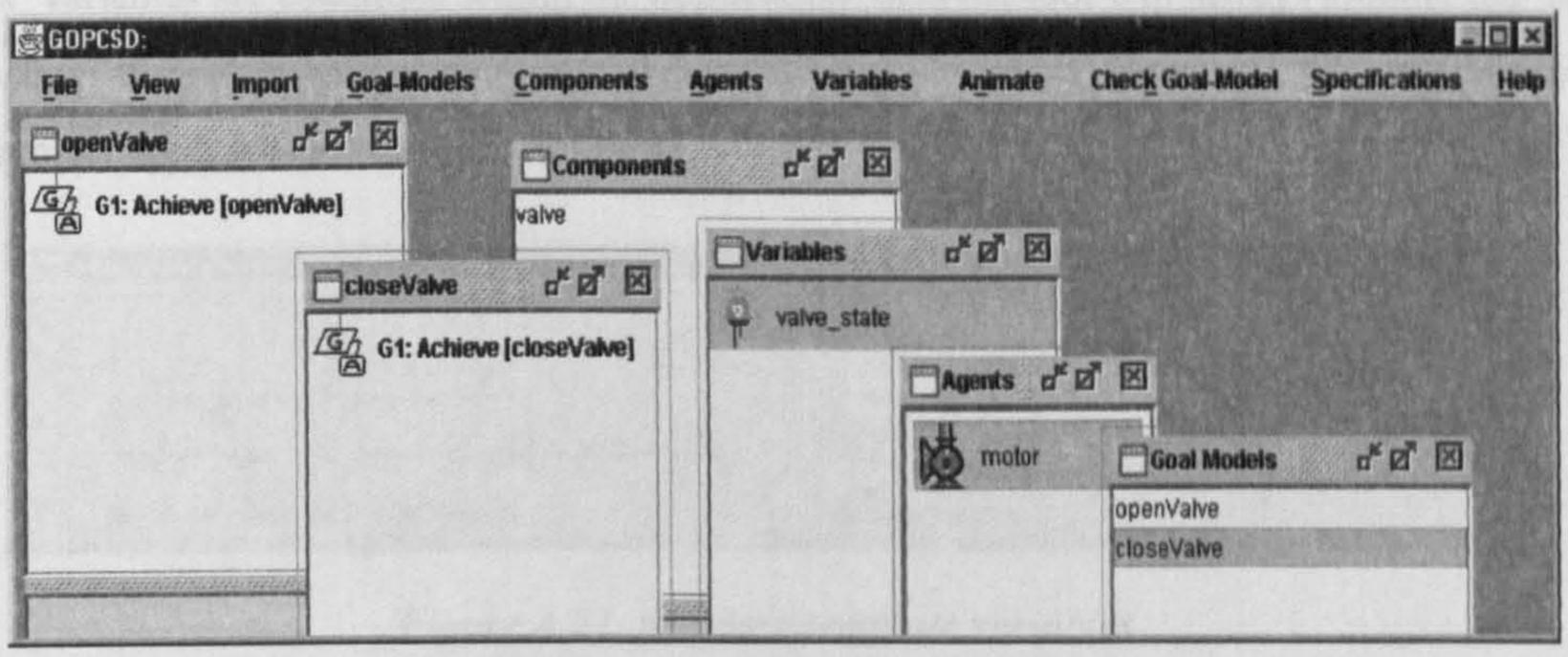


Figure A.19, the application after importing the valve component

A.4.2.3 Importing High-level goal-model Templates

Normally, after the user imports all the application components, he can import the high-level goal-models that are the templates for the goal-models. These templates can be used to combine the components' low-level goal-models and the other goal-models that the user can add later to construct the complete application goal model.

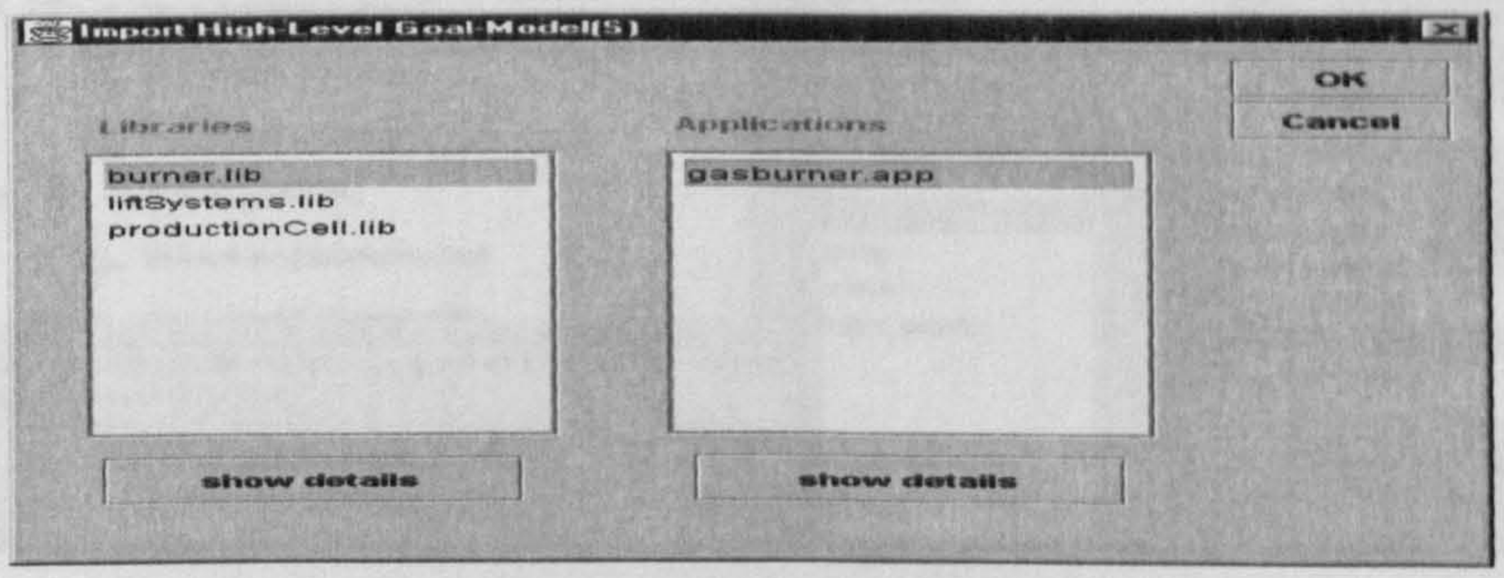


Figure A.20, Importing high-level goal-model

A.4.2.3.1 Selecting the High-level Templates

The user uses the Import/High-level goal-model sub-menu to display the import high-level goal-model Dialogue Box that can be used to import goal-model templates, as shown in figure A.20.

Unlike components, templates do not have specific names and they will all be combined into a single goal-model. Thus, the following step is to map the templates' variables and agents to the existing applications' variables and agents, respectively.

A.4.2.3.2 Renaming the Template Variables

The dialogue box in figure A.21 appears after the user imports high-level goal-model templates from the library. The high-level goal-model usually contains the higher-level functions expected from such an application, but does not have the full details of how to achieve them, which mainly depend on the components' details that can differ from one application to another. We argue that this template concept can increase the overall flexibility of the tool. The dialogue in figure A.21 is the same as the dialogue for mapping the variables of the components.

However, if the user imported the components before importing the templates, usually the templates' variables are contained within the application, thus the tool will always assume that the user needs to map the template variables into the application variables; however, the user can add the variables to the application if they are not there.

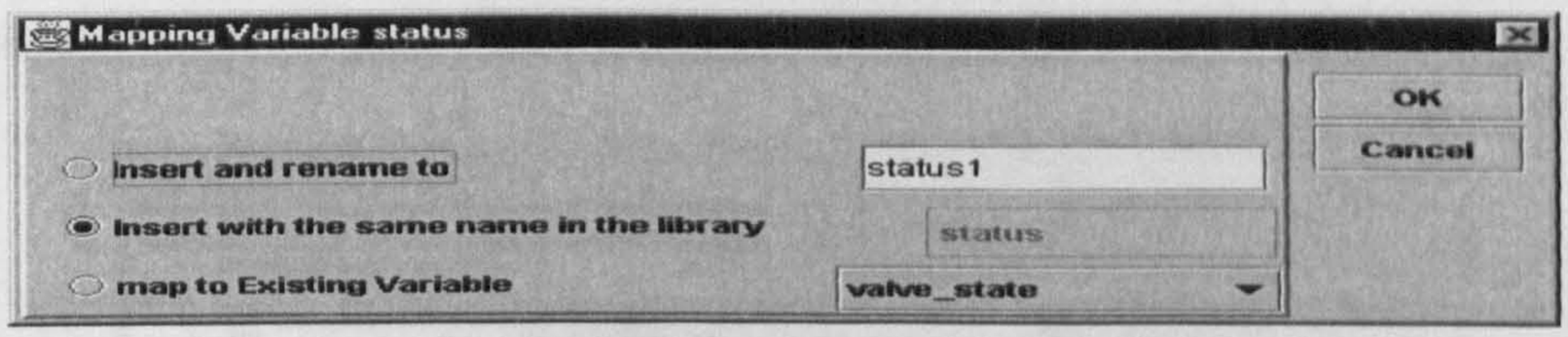


Figure A.21, mapping template variables

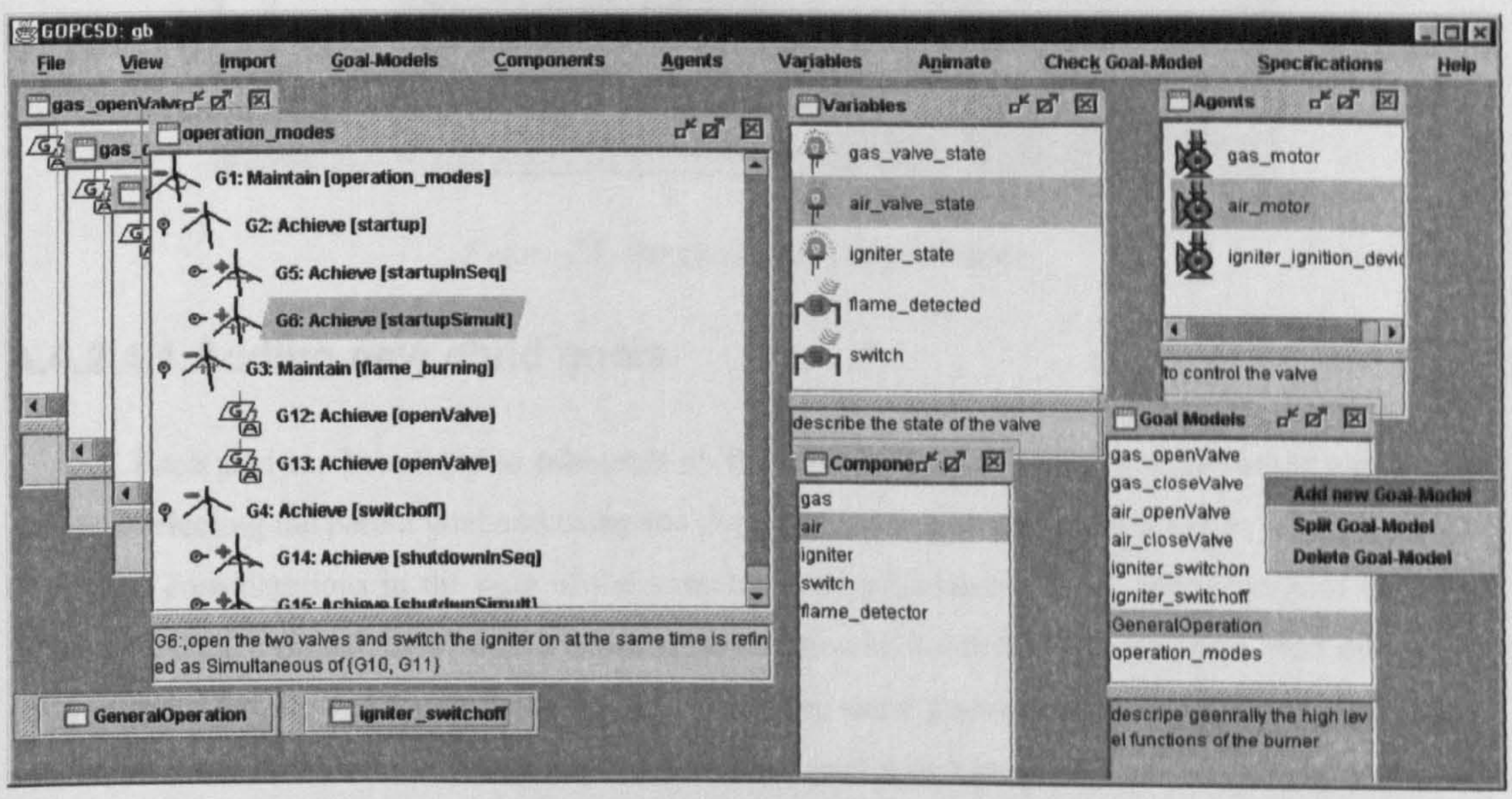


Figure A.22, the GOPCSD after importing the relevant components and templates

A.4.2.3.3 Renaming the Template Agents

The agent mapping dialogue for the high-level templates is the similar to the corresponding one of the components. However, it less probable to find templates containing agents because templates usually specify high-level goal-models, whereas agents are found at the level of the terminal goals to accomplish components' low-level terminal goals.

Figure A.22 shows the GOPCSD desktop after importing the appropriate components and high-level goal-model templates. The goal-models list to the right contains the names of the goal-models found to the left. The goal-model that describes the operation modes of the gas burner is shown named operation_modes. It has different types of goals organised in a tree hierarchy.

After describing how to build the skeleton of the application by importing the components and the templates, we describe how to edit the goals (the building blocks of the goal-models).

A.4.2.4 Goal-model popup menu

Each goal-model enables the user to manipulate its goals and the entire goal-model using a popup menu as shown in figure A.23. Each goal inside the goal model is uniquely identified by an index, which is an integer identifier appearing in front of the goal name and is automatically updated each time the user changes the goal-model structure by introducing, removing or moving goals, as will be explained in the following sections.

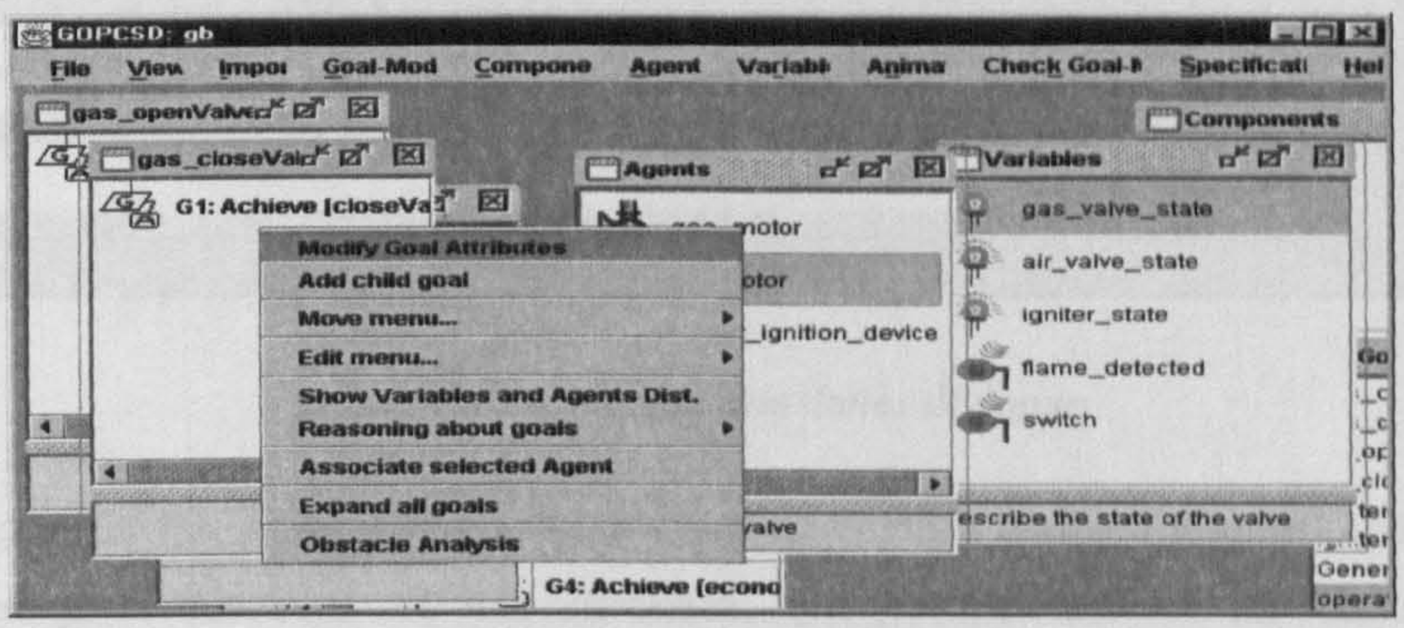


Figure 23, the goal-model popup menu

A.4.2.4.1 Adding new child goals

Each goal can be refined to sub-goals by first deciding a refinement pattern for the parent goal and then selecting the parent goal and using the popup menu to add new child goals to it. There should be special considerations in the case of the simultaneous refinement pattern, where each of the child goals should be terminal; also, in the inheritance refinement, by definition, not more than one single child goal can be allowed for each parent goal within the same goal-model. After the user clicks add a new child, the goal-model will be updated and the new goal will appear with new-child-goal name and default attributes that normally requires the user to change the attributes' values as explained in section 4.2.4.2.

A.4.2.4.2 Editing Goal Attributes

When the goal-models are imported from the library or created for the first time using the dialogue box in figure A.11, they contain goals that may require attribute modifications; as the user selects one of the goals inside the goal-model, a brief description appears inside the goal-model frame. But, if the user wants to see the goal's details and to manipulate them, he can activate the popup menu

using the mouse right button and select edit attributes. Figure A.24 shows the goal dialogue where the user can edit goal attributes like name, informal description, and goal type, and decide whether the goal is terminal or not.

For the formal description of the goal, the tool enables the user to use a predictive parser that can guide the user on what to write, as shown in figure A.25. The user can click the formal button in the goal dialogue in front of the formal description of the goal condition and action, and the formal description dialogue will then pop up.

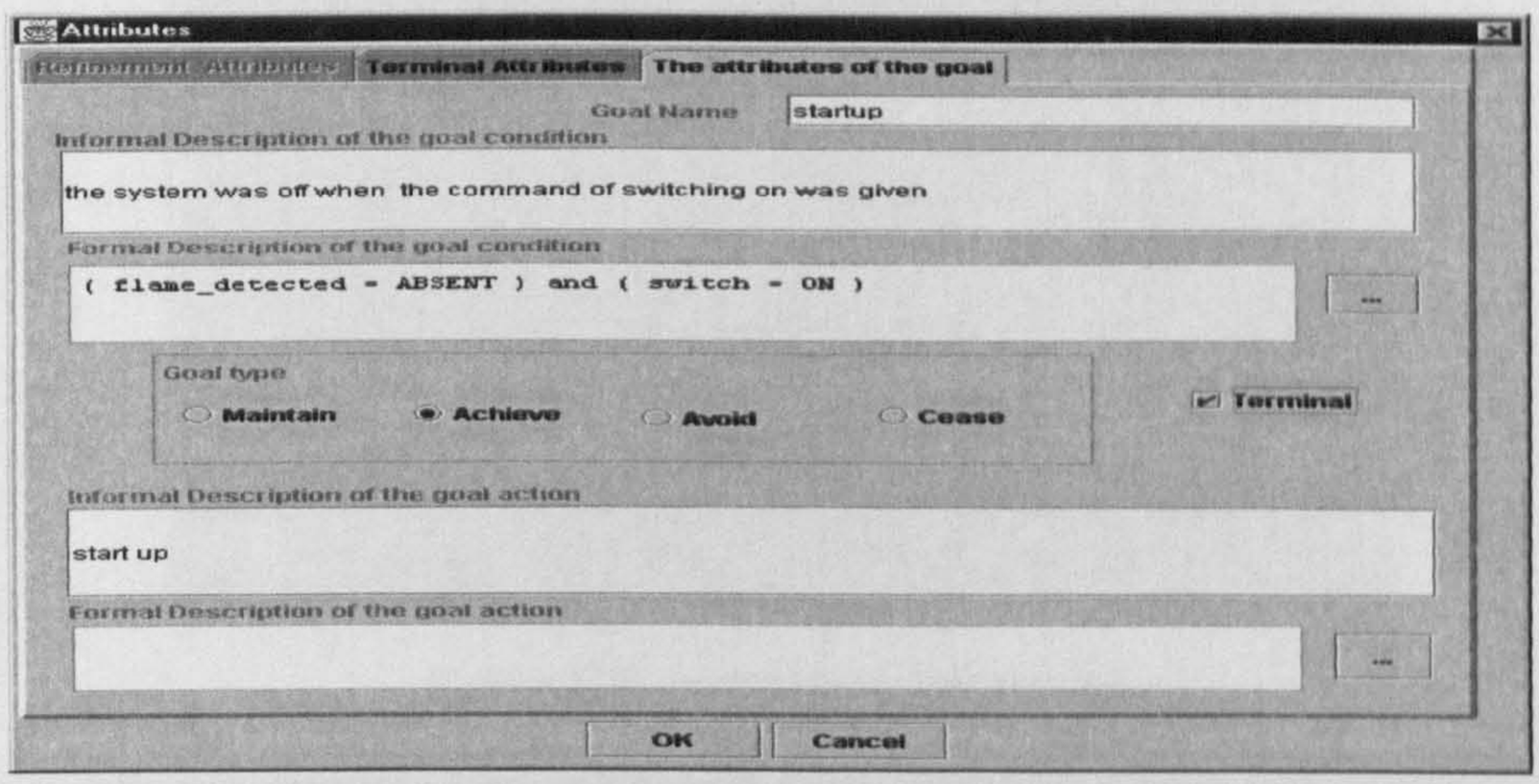


Figure A.24, the goal attributes dialogue

In figure 24 the dialogue has a checkbox to distinguish between terminal and non-terminal goals. According to the state of this checkbox, one of the tabs of the dialogue terminal attributes/refinement (for non-terminal goals) attributes will be active and the other will be inactive.

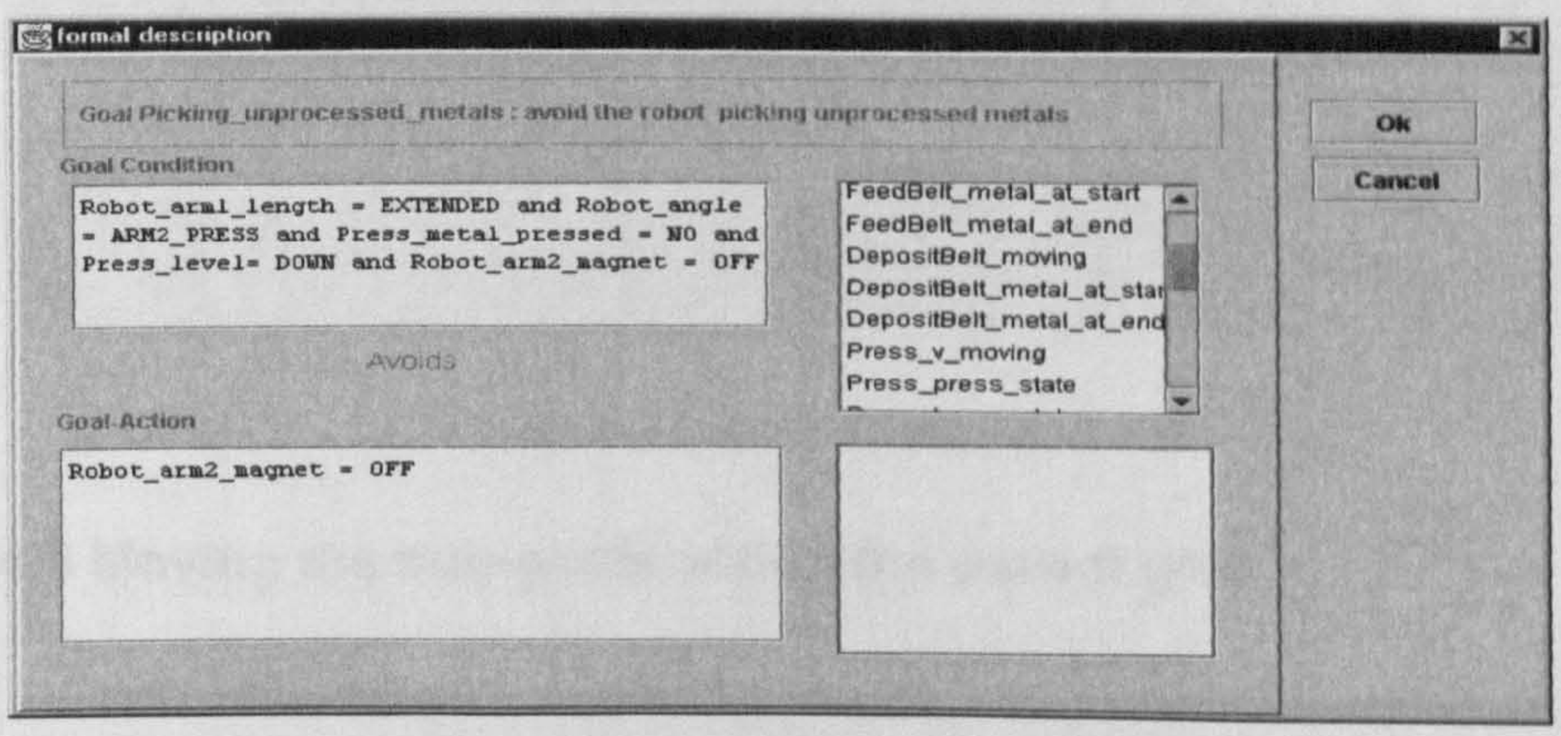


Figure A.25, the formal description dialogue

A.4.2.4.2.1 Terminal goals

In figure A.26, the terminal attributes tab of the goal dialogue appears to enable the user to select the type of the terminal goal, whether it is environmental or not and whether it is functional or

not. Environmental goals can appear during the animation and conflict checks but not in the generated specifications because they are part of the external environment.

A.4.2.4.2.2 Non-terminal goals

In figure A.27, the non-terminal attributes tab of the goal dialogue appears to enable the user to select one of the refinement patterns for the sub-goals of the edited goal. Each pattern is followed by a brief description for the user's guidance.

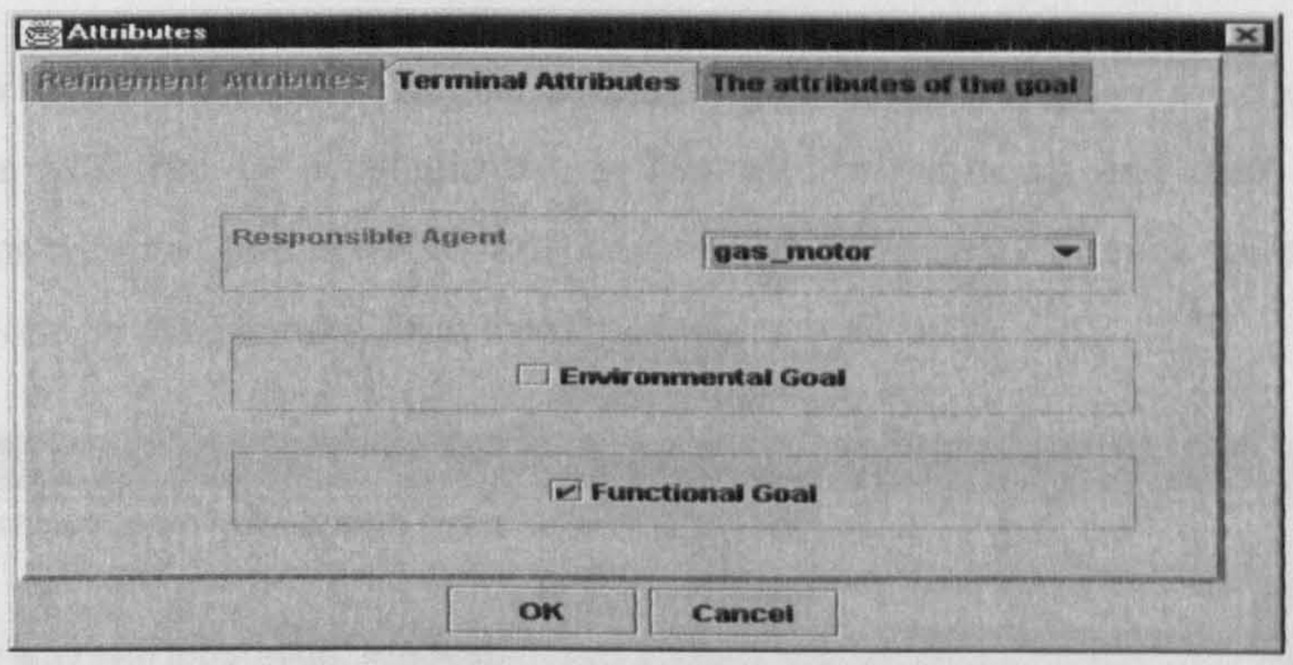


Figure A.26, the terminal attributes tab

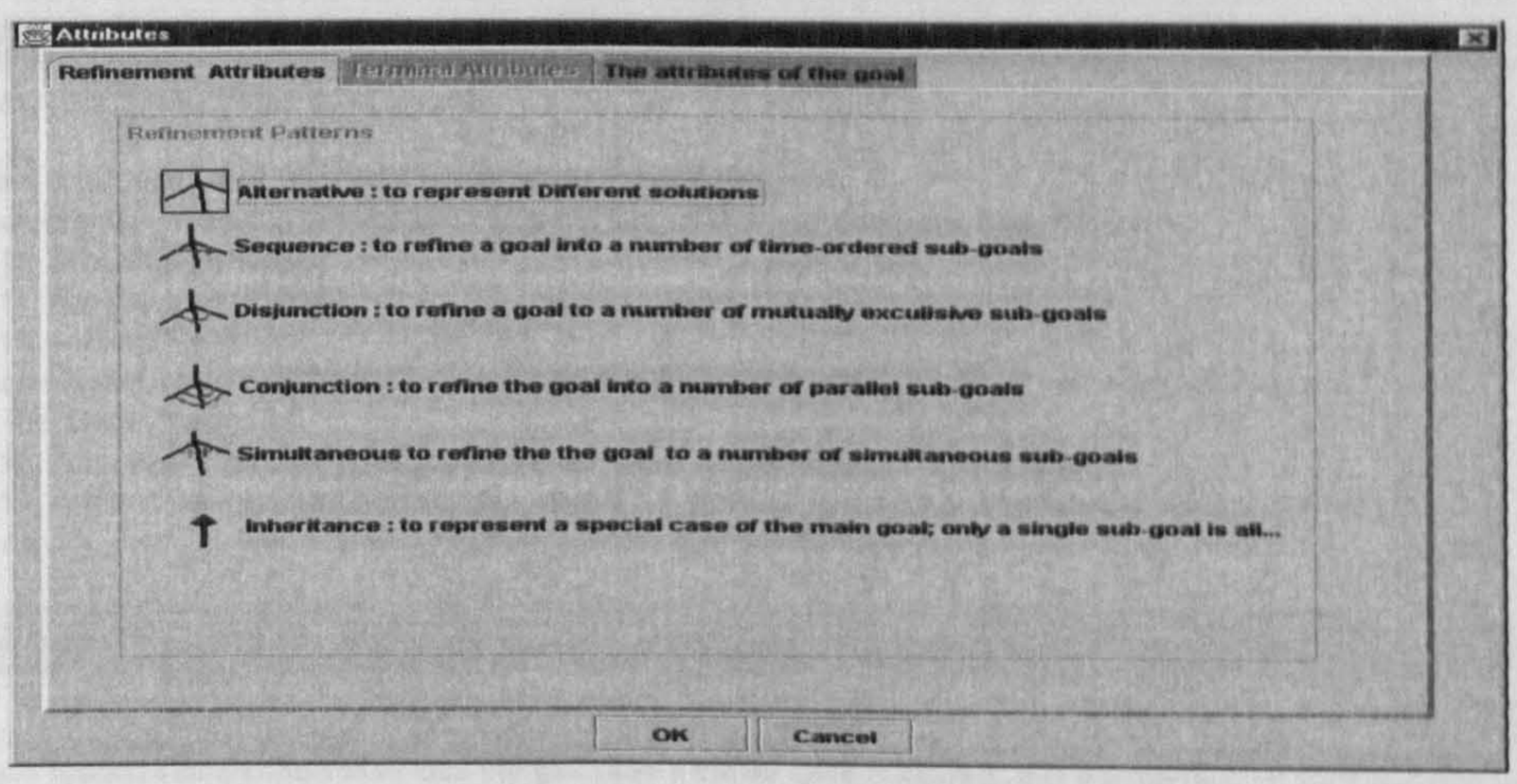


Figure A.27, the refinement tab

A.4.2.4.3 Moving the sub-goals within the parent goal

The tool enables the user to re-order the sub-goals using a standard move first, next, previous, and last; this option can be activated through the popup menu and then selecting move goal.

A.4.2.4.4 Copying, Cutting, Pasting, Deleting goals

The goal-model popup menu enables the user to duplicate, delete, move goals (move it outside the scope of its parent using sequence of cut then past) and sub-goal trees by selecting the desired goal, then activating the popup and selecting the edit sub-menu followed by cut, copy, delete

etc. in a standard way, like most applications. It should be noticed that the paste option works either to add the clipboard goal as a new child for the selected goal (paste into new child) or to replace the whole goal by the clipboard goal (paste into). Being able to copy and paste between different goal-models, the user can combine different goal-models or sub-parts through creating a new goal-model and then copying each goal-model root or desired goal, and then pasting it as a new child in the new goal-model.

A.4.2.4.5 Reasoning about the goals

The user can reason about each goal within the goal model using how or why; how reasoning explains how the goal can be accomplished; it lists all the sub-goals and their sub-goals. Why reasoning explains why the goal needs to be accomplished; it traverses through the ancestor goals of the selected goal one by one, starting from the direct parent goal.

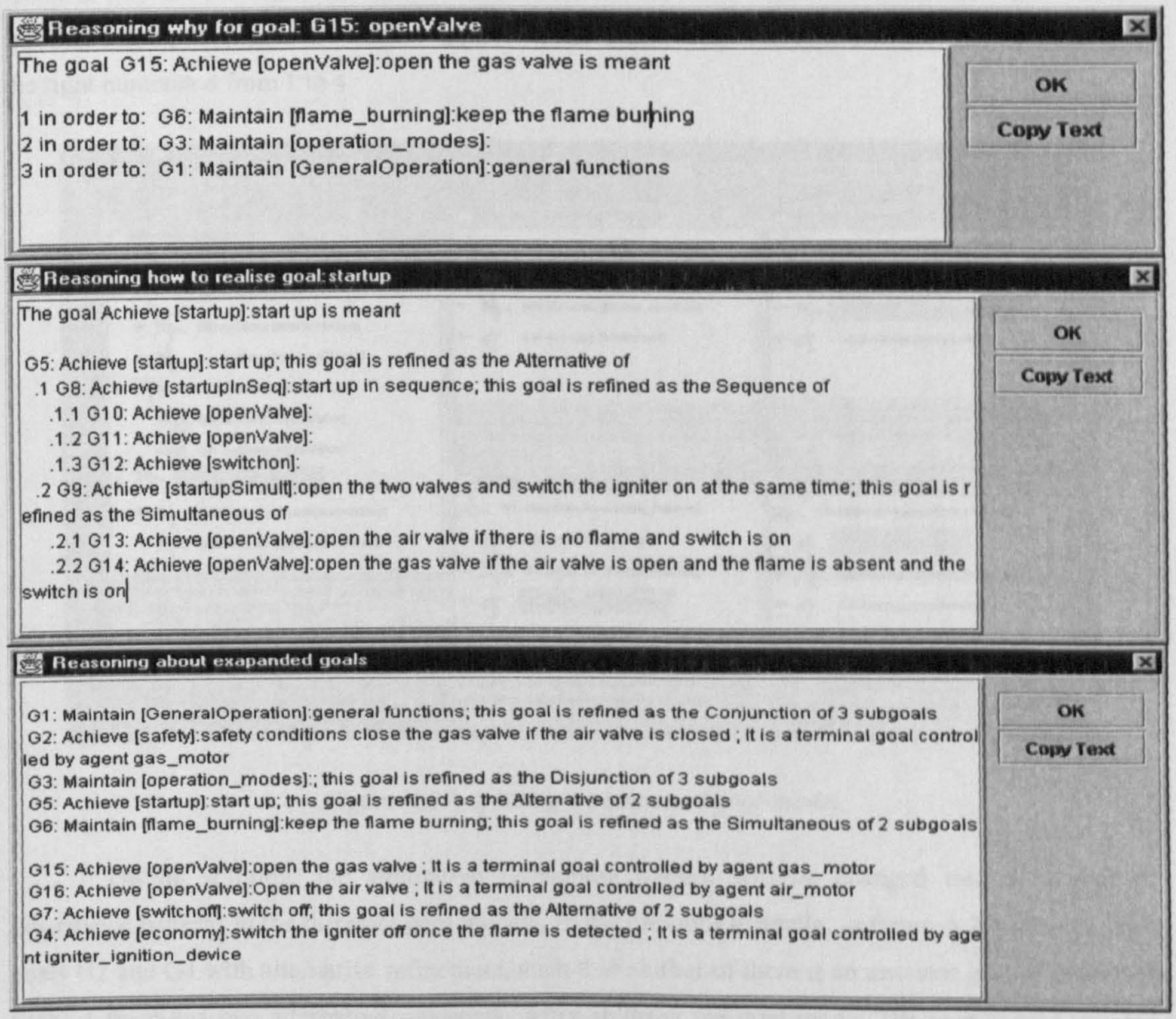


Figure A.28, reasoning about goals.

Moreover the user can expand the most important goals and collapse the rest then a detailed report can be displayed for him/her when he/she selects the reason/ reason about expanded goals sub-menu. Figure A.28 shows the message boxes that pop up as a result of different goal reasoning.

A.4.2.4.6 Expanding the goal-model

The tool enables the user to expand the goal-model tree directly through the popup menu expand all goals. This can help the user to see all the details of all goals. We have not mentioned the obstacle analysis menu item that appears in the goal-model popup menu yet, because we thought it is more relevant to describe it in the second phase (section 4.3.5)

A.4.2.5 Splitting the goal-model

The complete goal-model, which the user constructs usually, contains alternative refinement sites; these refinement sites mean a set of integrated solutions. This solution-integration reduces the user's effort and shortens the time required to construct the separate goal-models; however, a single version is the target. Thus, the compound goal-model needs to be split into simple solutions by splitting the alternative goals at each alternative refinement site. Figure A.29 shows an example of splitting a compound goal model that appears to the left. The resultant simple goal-models appear to the right numerated from 1 to 4.

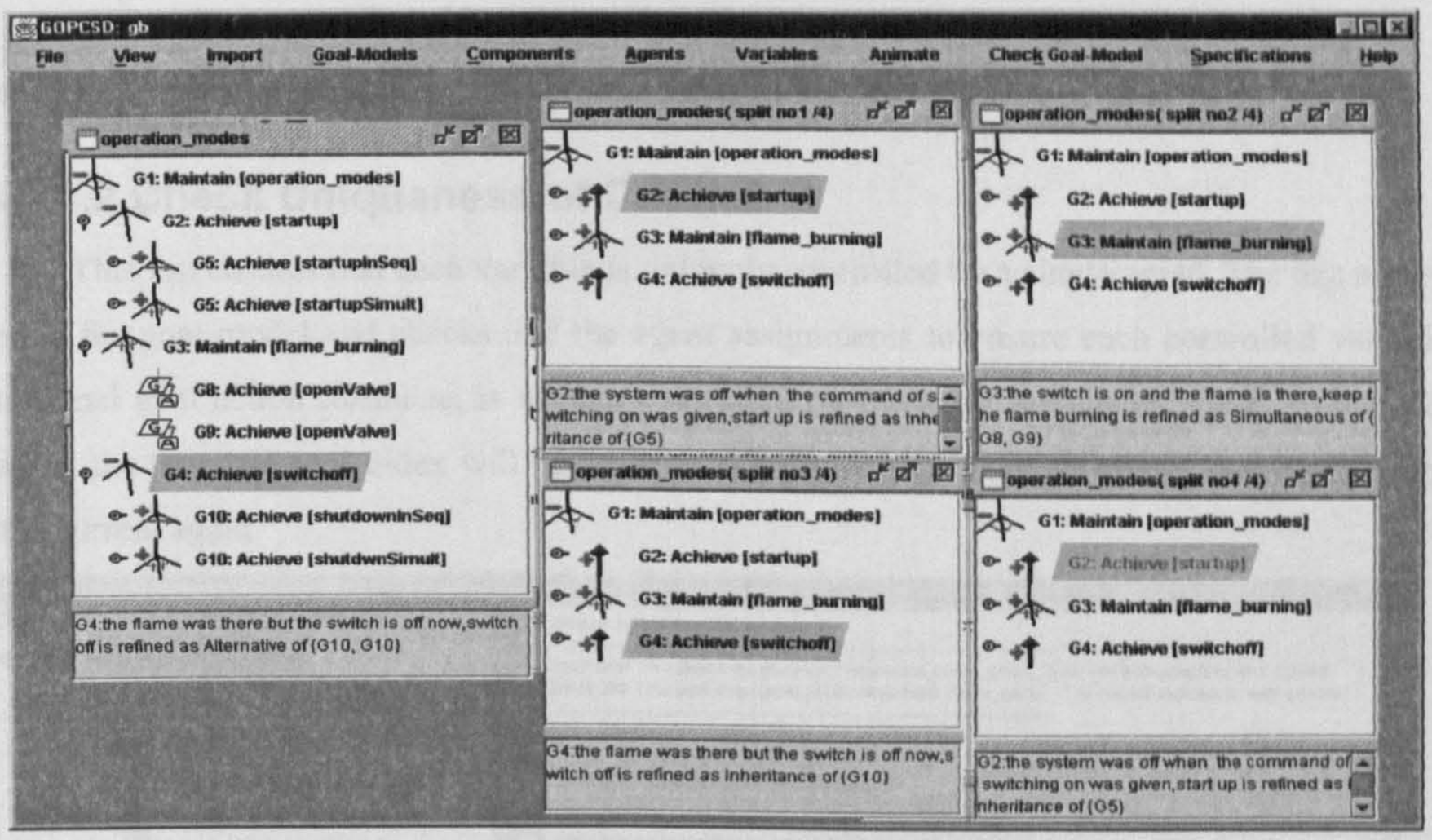


Figure A.29, splitting a compound goal-model

During splitting, the alternative refinement pattern will be changed into a number of inheritance patterns in the different goal models. In the previous example, in figure A.29, there are two goals G2 and G4 with alternative refinement, such that neither of them is an ancestor goal of the other; each of them has two alternative sub-goals. After splitting the goal-model, the user should proceed with the simple goal-models within the second phase and modify each of them according to the feedback from the different tests, as will be explained in section A.4.3.

A.4.3 Phase II, Checking the goal-models

The tool employs phase one and two as a feedback loop for the user to provide him suggestions to modify the requirements. In the second phase, the tool provides a general check for the goal-model to ensure the basic restrictions it has to fulfil, conflict analysis to detect inconsistency, completeness, goal-reachability, and, finally, obstacle analysis to predict problems that can occur during run-time and prevent the application from operating correctly as planned. Moreover, the tool enables the user to validate the requirements and correct the logical errors.

A.4.3.1 Checking the basic goal-model structure constraints

The user can check the fulfilment of the basic constraints in the developed goal-models using the menu item Check/Check goal-model; hence, the dialogue box in figure A.30 will pop up and show the result of the named tests. Moreover, the tool highlights the goals violating the goal-model constraints to help the user locate them easily.

A.4.3.1.1 Check Agent Assignments

This test ensures that every terminal goal is assigned to an agent. In the case there is a terminal goal without associated agent, the goal index will be reported.

A.4.3.1.2 Check Uniqueness of Control

This test ensures that each variable is uniquely controlled by a single agent. The test algorithm traverses the goal-model and checks and the agent assignments to ensure each controlled variable, in the terminal goal action formulae, is always controlled by the same associated agent. In the case of violation, the terminal goal index will be reported, along with the variable name, the expected agent, and the current agent.

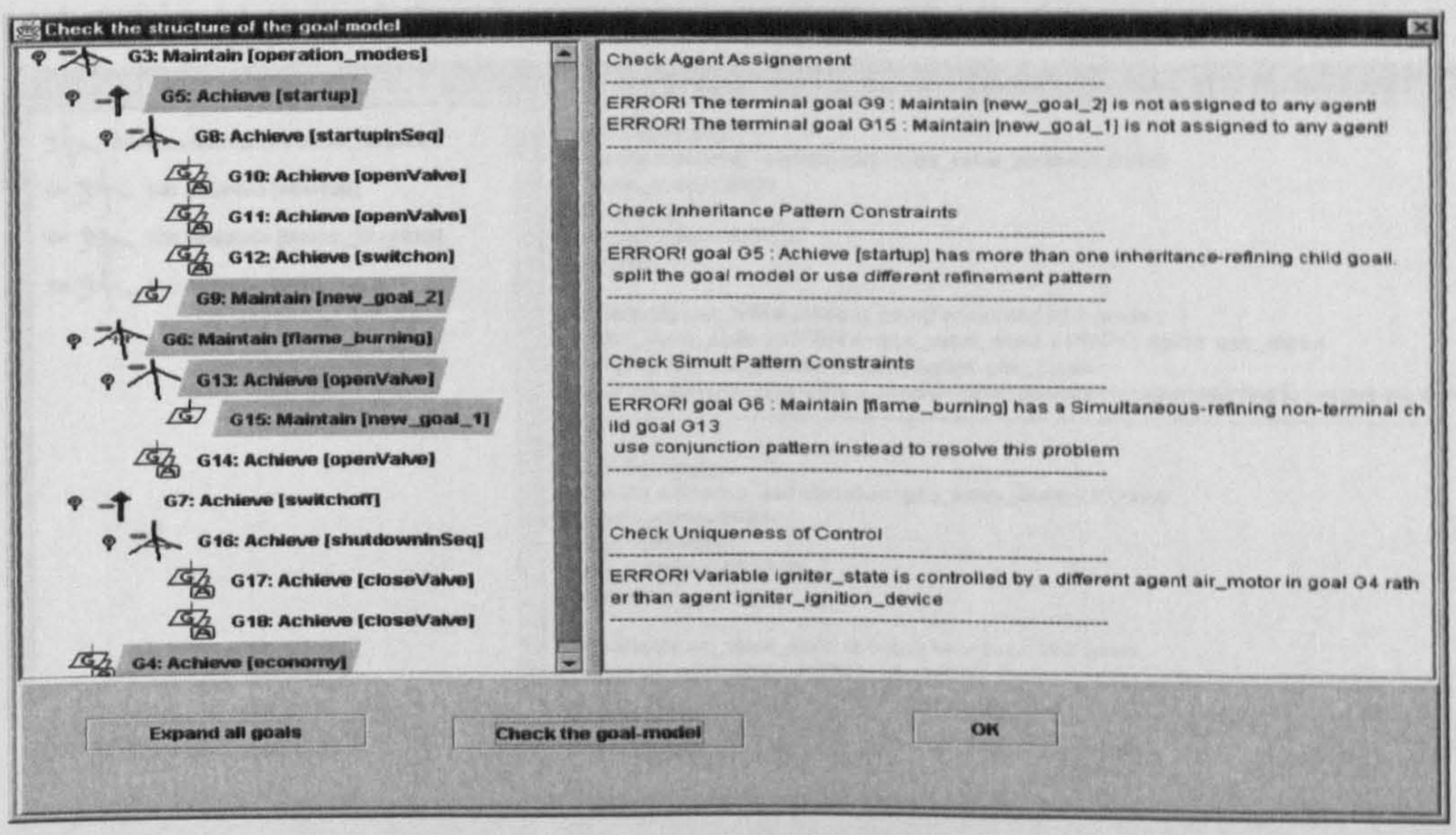


Figure A.30, Check the goal-model basic constraints

A.4.3.1.3 Check Inheritance refinement Constraints

This test ensures that each inheritance refinement site has exactly a single goal. The inheritance pattern means that, under all circumstances, the child goal is a special case of the parent goal; thus, it will be inconsistent if two instances exist in the same single solution version.

A.4.3.1.4 Check Simultaneous refinement Constraints

This check ensures that each simultaneous refinement site has only sub-goals of type terminals. In case of having a non-terminal sub-goal, the user will be guided to change the refinement pattern to conjunction; or alternatively, he can change the goal-model structure.

A.4.3.1.5 Alternative refinement constraints

Unlike the constraints of the simultaneous and inheritance refinement pattern, the alternative constraints is not an error but a kind of warning to inform the user that this goal-model needs splitting before it can proceed to the following check stages. The algorithm basically traverses the goal-model tree and reports any non-terminal goal that has alternative refinement type.

A.4.3.2 Check Goal-Conflict

Consistency is a crucial aspect of the application requirements. Although the developed application usually seems consistent from local view, under some critical circumstances some parts of the application can prescribe inconsistent behaviour [Easterbook 94, VanLamsweerde 98b]. This may lead to critical and unexpected hazards or at least improper operation. The GOPCSD tool enables the user to check the consistency of the goal-model, as shown in figure A.31. The user can perform this check on the goal-model to discover the conflict goal pairs. Usually, the conflict happens when two goals try to control the same variable simultaneously. The user can activate this check by selecting the goal model to be checked, then clicking on the goal-model check/goal conflict sub–menu.

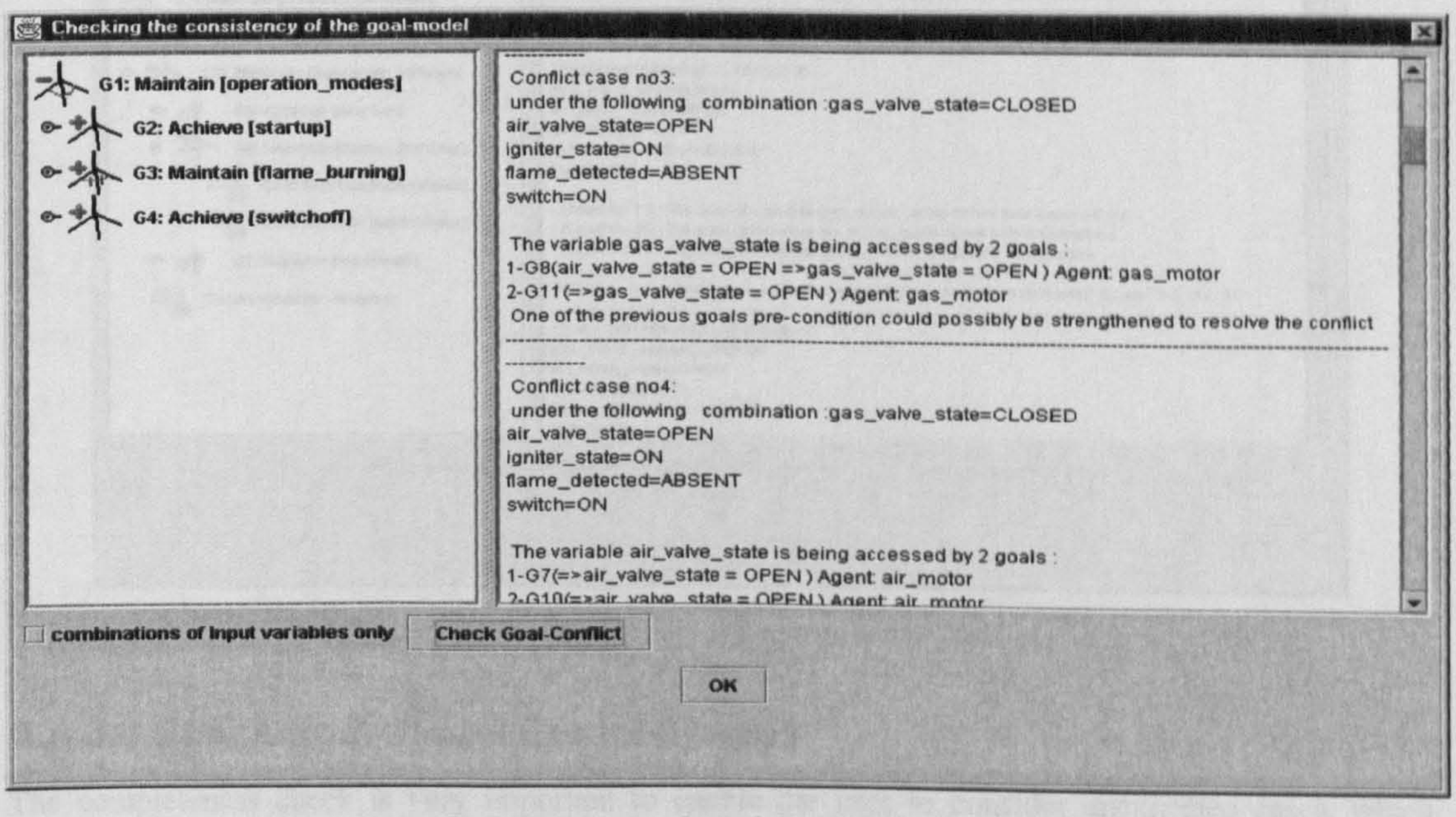


Figure A.31, goal-model validity check.

A.4.3.3 Check goal-reachability

Figure A.32 shows a dialogue box that report unreachable goals. Sometimes, especially in large applications, the user is more likely to err in specifying the pre- or post- condition of one of the goals; this may result in a case where this goal will not be active at any time. Thus, the tool discovers such cases and reports them to the user with some suggestion and possible predictions about why these goals are unreachable.

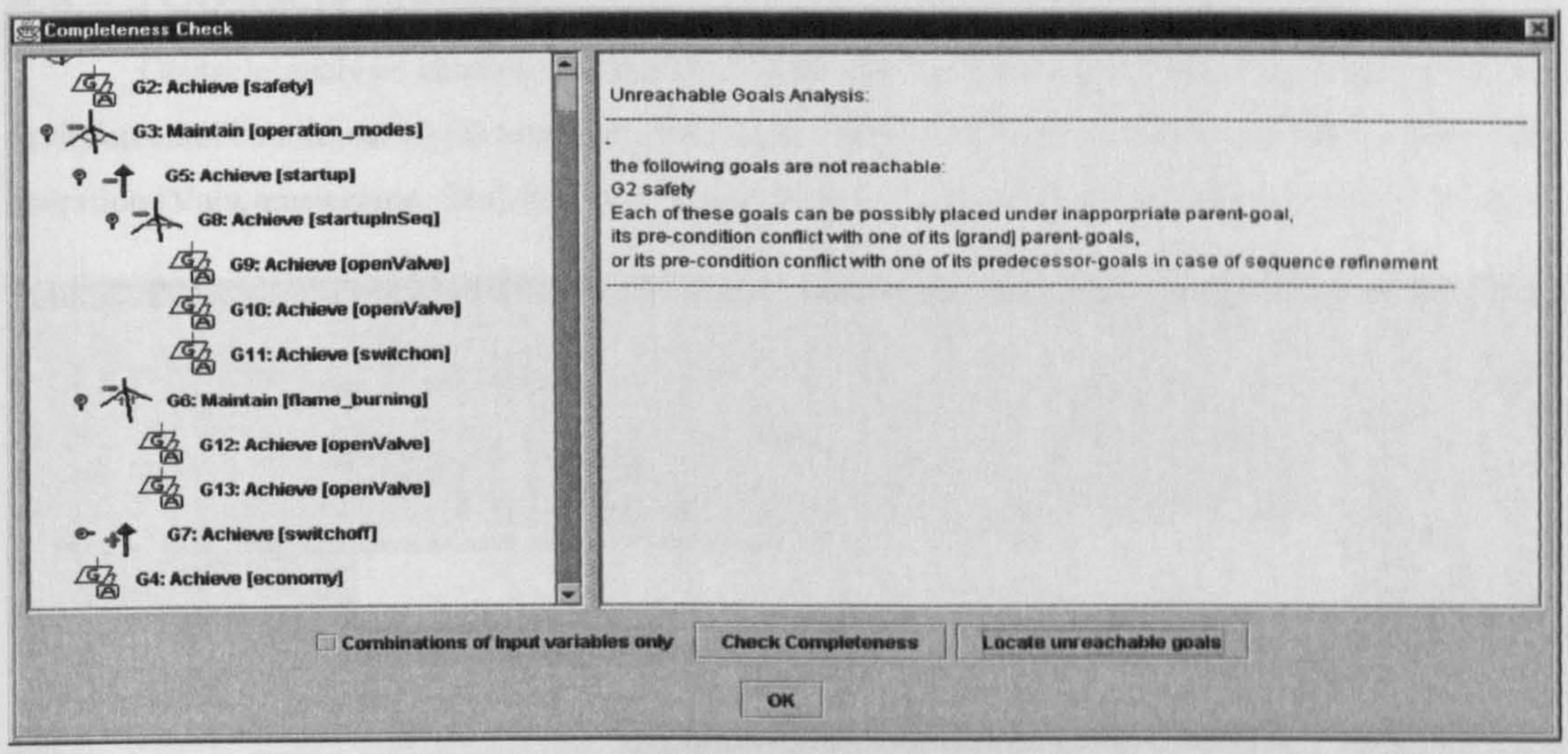


Figure A.32, checking whether each goal is reachable or not

It is important to know that having an unreachable parent goal results in an unreachable sub-goal. This test can be activated by selecting the desired goal-model, then clicking on the check goal-model/completeness and reachability check sub-menu and then clicking the locate unreachable goals button as shown in figure A.32.

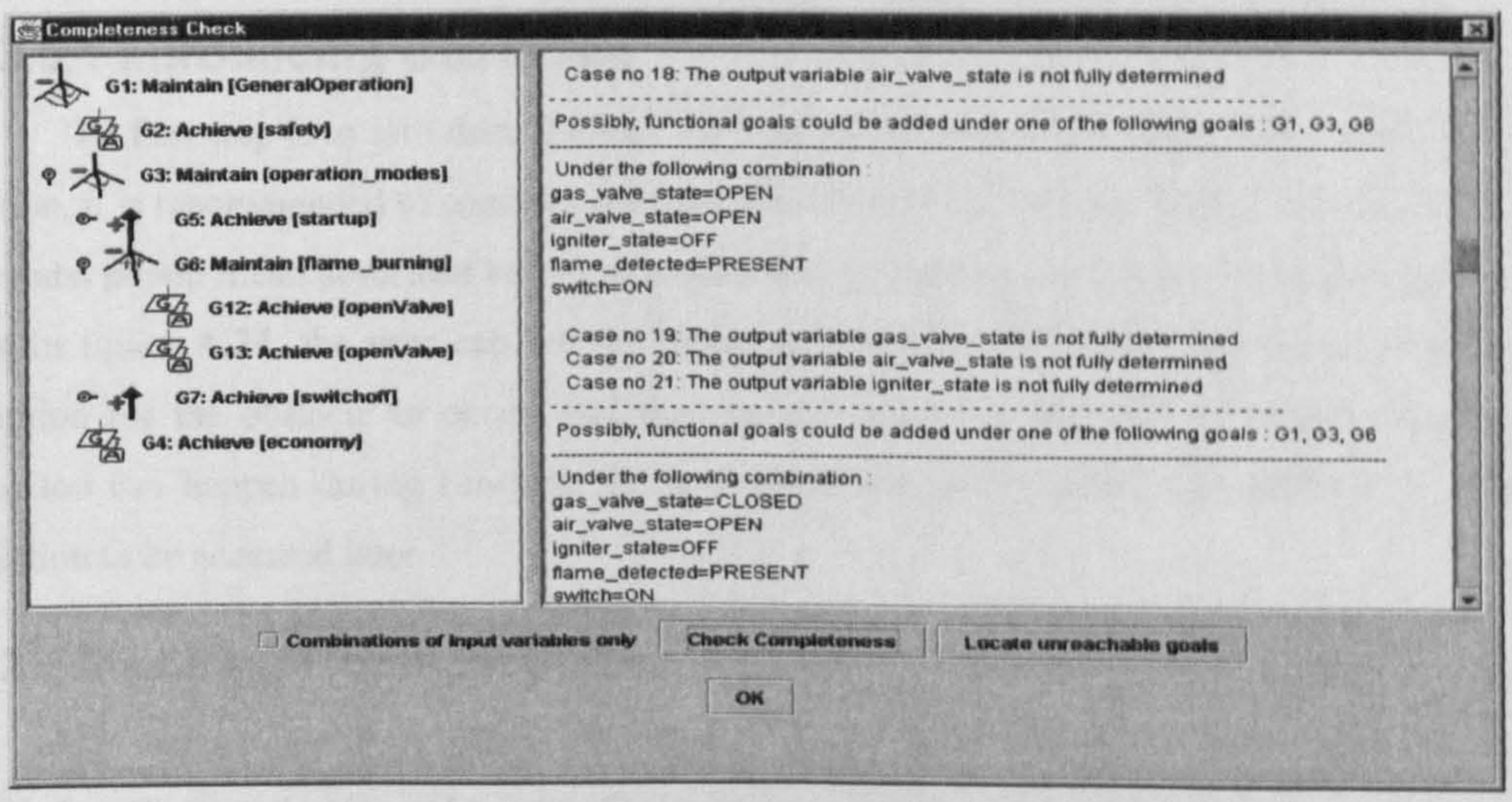


Figure A.33, check the completeness of the goal-model

A.4.3.4 Check goal-model Completeness

The completeness check is very important to enable the user to consider unexpected cases, which he/she did not consider. The test can be activated by selecting the desired goal-model, then clicking on

the check goal-model/completeness and reach-ability check sub-menu and then clicking the check completeness button, as shown in figure A.33.

The tool should provide enough feedback in the case of discovering incompleteness; hence, the user can either complete the goal-model by adding covering goals, weakening the pre-conditions of goals, or just be aware of this incompleteness if he knows such a situation is impossible.

A.4.3.5 Obstacle Analysis

Obstacle analysis enables the user to deepen the application's scope by adding/considering environmental conditions which were not considered before and which usually affect the application's operation [VanLamsweerde, 98a]. The user can perform the obstacle analysis in three steps as follows:

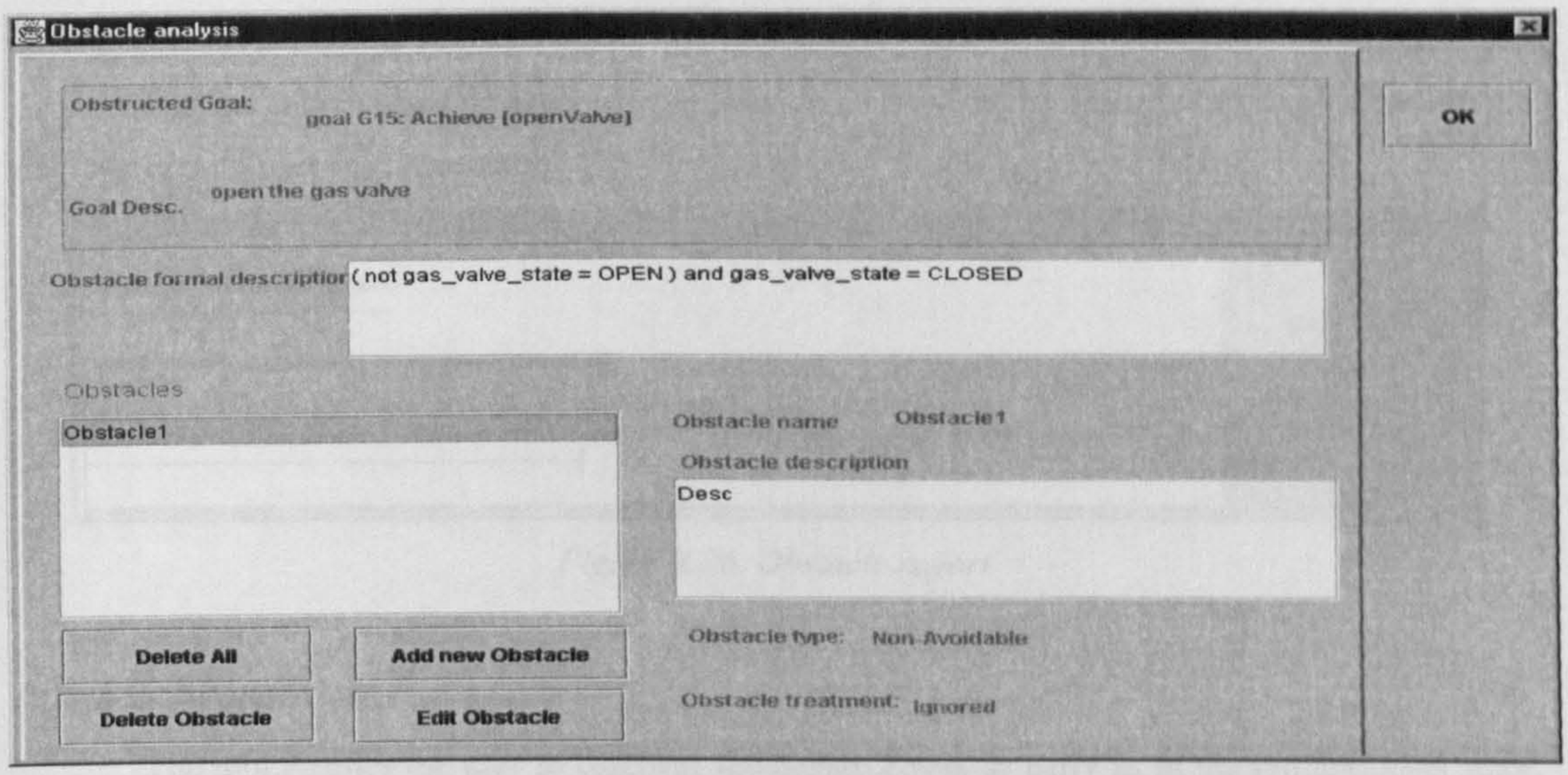


Figure A.34, adding obstacle to a specific goal

A.4.3.5.1 Introducing Obstacles

The first step is to introduce the tool via obstacles that obstruct some of the goals. For effort reduction, it is recommended to consider only the obstacles of the terminal goals. The user can use the goal-model popup menu described before in section 4.2.4 to activate the obstacle analysis dialogue. As shown in figure A.34, the user can see the negation of the goal's formal description, which is the description for the obstacle to occur, and then he/she can think of different reasons why such an obstruction can happen during run-time. Each obstacle will be recorded with different a name and description to be accessed later.

A.4.3.5.2 Editing Obstacle Status

The second step is to change the obstacle status (user response to it) by first selecting the desired obstacle from the Obstacles list in figure A.34, and then clicking on the edit obstacle button.

Each obstacle can be avoidable, amendable, non-avoidable or removable. The GOPCSD tool provides some guidance to the user to inform him/her how to get rid of/avoid/amend the effect of the obstacle. Accordingly, he should update what is the current response for this obstacle.

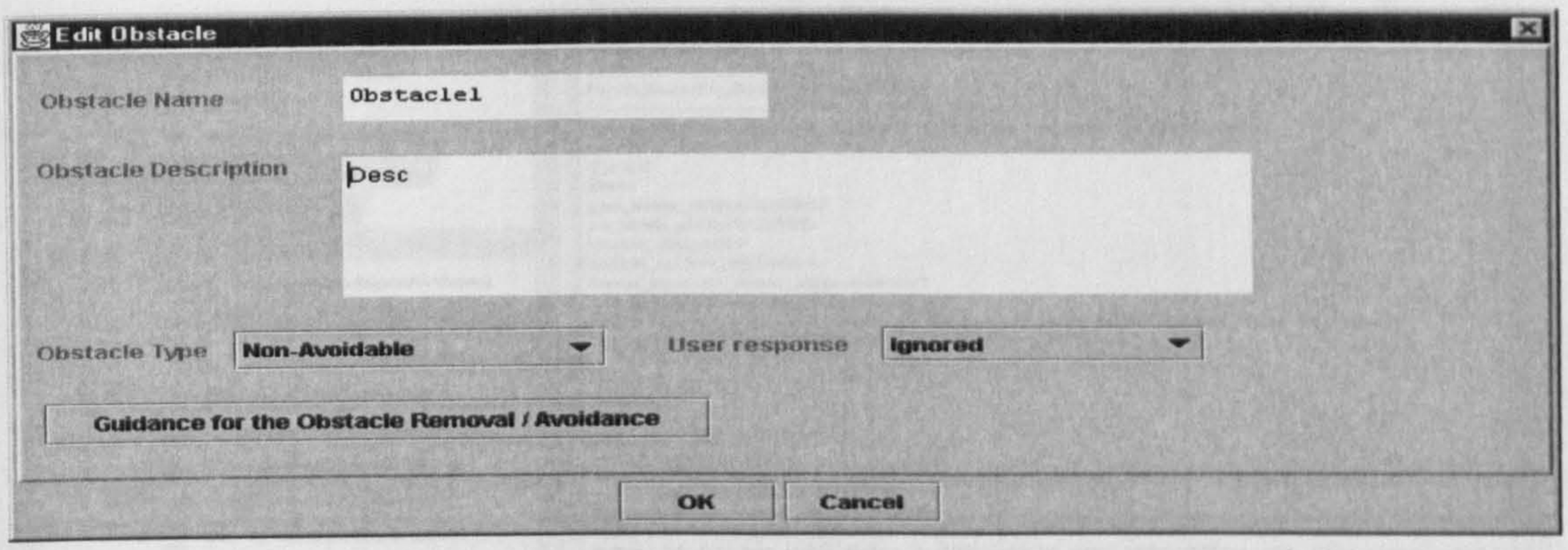


Figure A.35 modifying the obstacle status

A.4.3.5.3 Reporting Obstacles for the entire goal-model

The last step in obstacle analysis is to provide a report on the obstacles within the goal-model and their current user-response status.

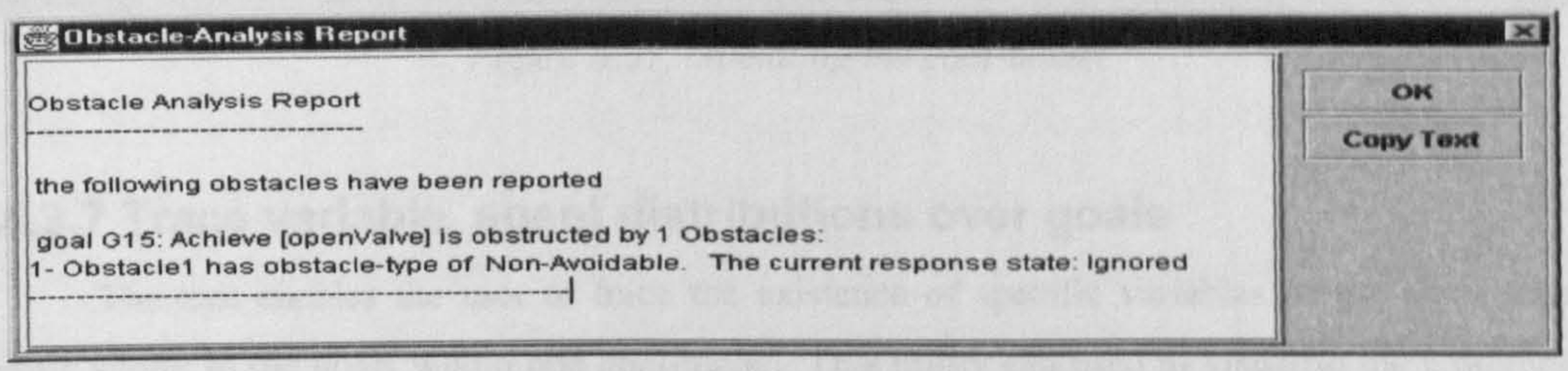


Figure A.36, Obstacle report

A.4.3.6 Animation

The tool enables the user to discover the design errors as early as in the requirements stage. This can reduce the effort and the cost and shorten the final product production time. These errors can be discovered by validating the goal-model, as shown in figure A.37. The user can animate the goal-model by selecting the animate/step by step sub-menu.

The GOPCSD tool enables the user to use the animation utility in various ways to fulfil various proposes, as follows:

- The user can validate the requirements and accepts the system behaviour.
- The animation result may be used to differentiate between two versions of the application.
- The user can reset the variables to their initial values and execute different sceneries by changing the input variables to simulate receiving he different events.

The user can adjust the values of the application variables and observe the system behaviour in a response to sudden/unexpected events or changes in the output variables; this can be used to study application response in the presence of faults.

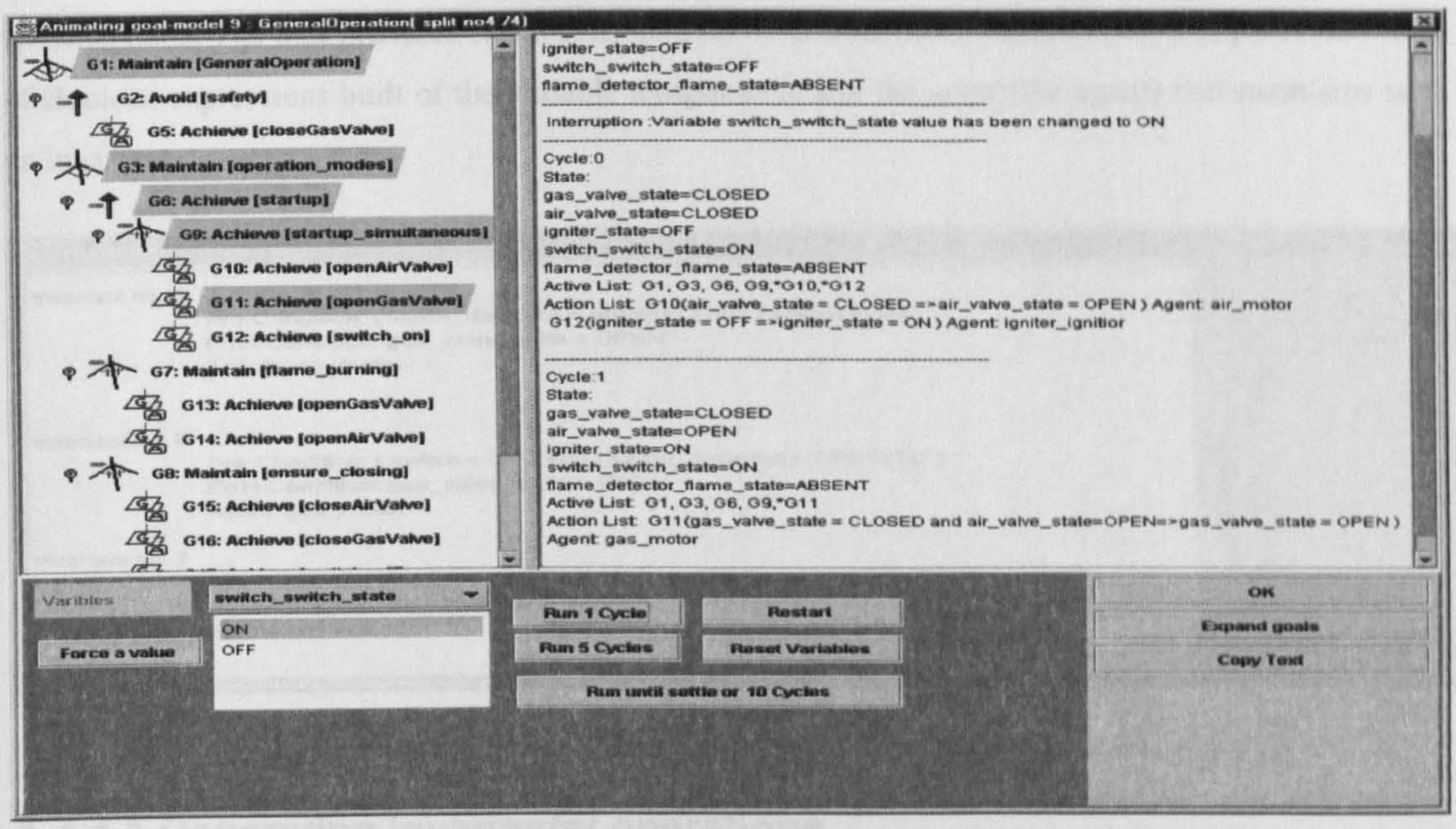


Figure A.37, animating the goal-model

A.4.3.7 Trace variable, agent distributions over goals

The tool enables the user to trace the existence of specific variables or the association of specific agents to the goals within one goal-model. This utility can used to visualise the requirements goals and to provide some help to trace some logical errors that could not be captured by the other checks and tests. Figure A.38 shows the dialogue box of the agent/ variables distribution over goals that can be activated from the goal-model popup menu.

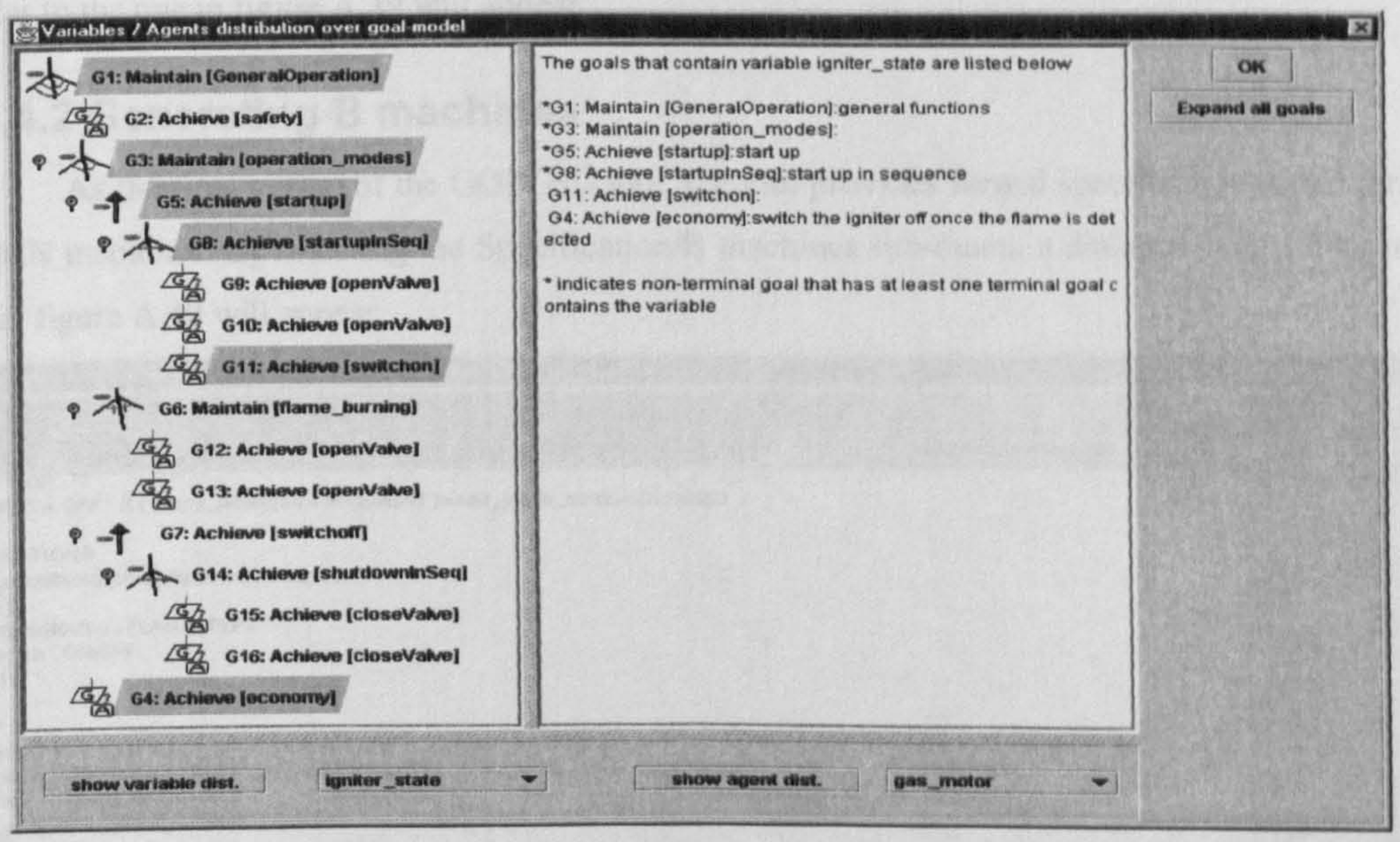


Figure A.38 locating goals containing one variable or associated with one agent

A.4.4 Phase III, Generate formal specifications

This phase is the final phase to automatically generate the output as a B specification. After the user checks and tests the application goal-model, the application will be ready for generating the

specifications. The tool provides two forms: general invariants that consist of pre- and post-conditions of Boolean expressions built of the variable assignments and the actor (the agent) that maintains such an invariant.

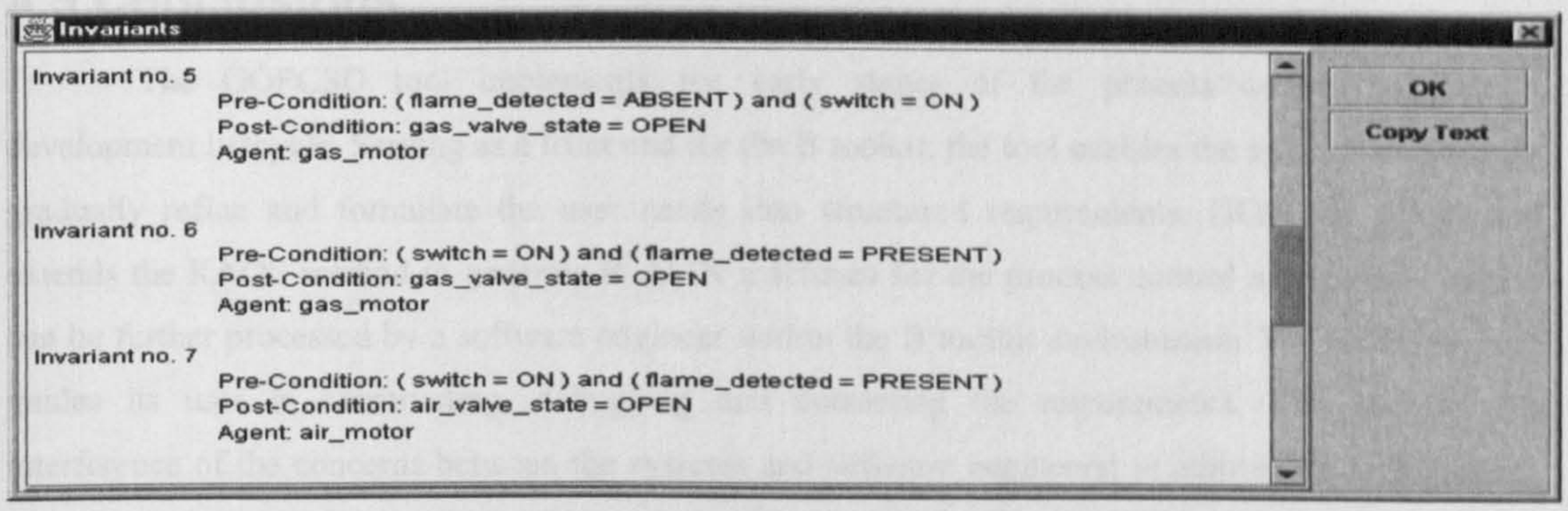


Figure A.39, the generated operational invariants/ operations

A.4.4.1 Generating invariants/ operations

In addition to the B specification the GOPCSD too generate operation and invariants, which can be used in other formal methods or directly into implementation programming languages. Invariants represent the logical input-output relationships between the application variables. The hierarchical structure of the goal-model is not shown in the invariants; only the functional and non-environmental terminal goals will contribute to the generated operations and invariants. These operations can be regarded as general specification units that can be the main input for other formal methods or programming languages. By selecting the Specification menu item/generate invariants, a dialogue box similar to the one in figure A.39 will appear.

A.4.4.2 Generating B machines

As the final output of the GOPCSD tool, the tool provides formal specification in the form of B AMN machines. By selecting the Specification/B machines sub-menu, a dialogue box similar to the one in figure A.40 will appear.

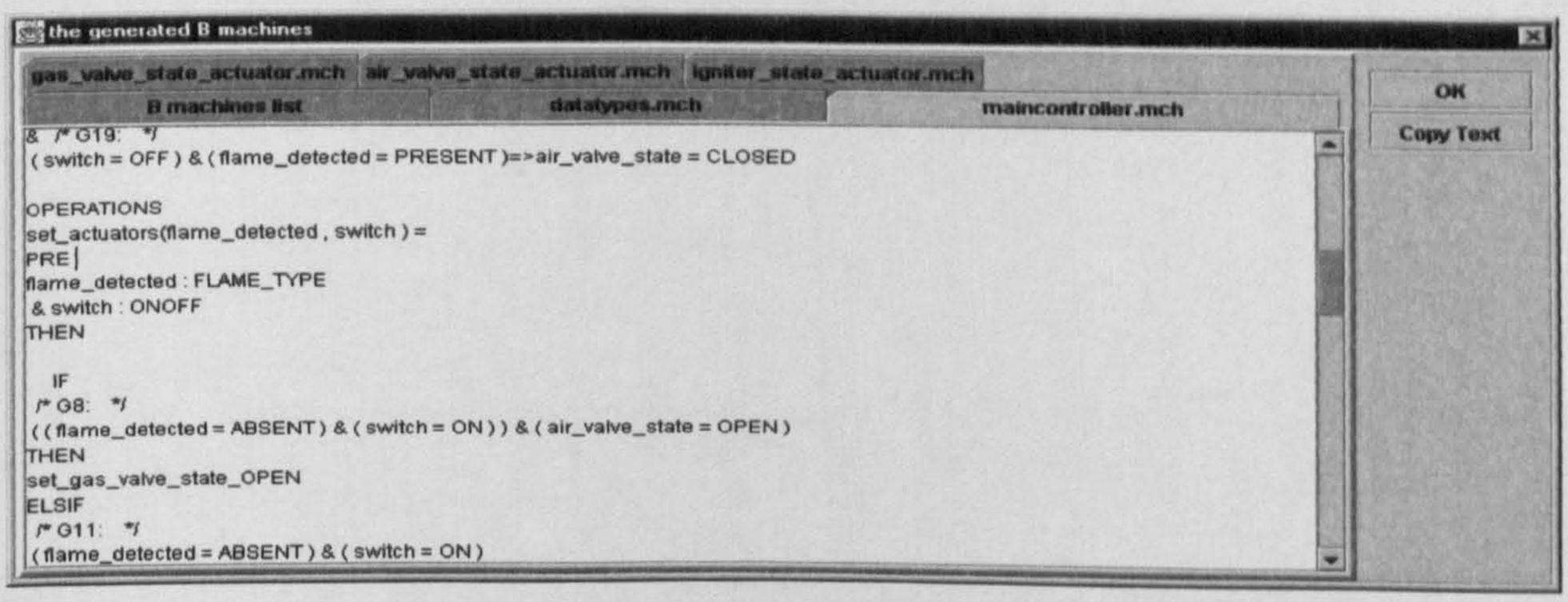


Figure A.40, the generated B machines

The tool provides a list of the generated B machines files inside the dialogue tabs; the user can browse the B files. The tool saves these specification files on the directory of the requirements application.

These B machines can be further refined and processed by the software engineer within the B toolkit environment with a high confident that the systems engineer agrees with their requirements.

A.5 Conclusions

The GOPCSD tool implements the early stages of the process control application development lifecycle. Serving as a front end for the B toolkit; the tool enables the systems engineer to gradually refine and formulate the user needs into structured requirements. GOPCSD adapts and extends the KAOS method to generate B AMN machines for the process control applications; which can be further processed by a software engineer within the B toolkit environment. The GOPCSD tool guides its user in constructing, debugging and correcting the requirements. This reduces the interference of the concerns between the systems and software engineers; in addition it decreases the overhead of the checking the formal specifications since the generated specifications have been examined and agreed by the user (systems engineer).

Appendix B

Details of the case studies

B.1 The Gas Burner System

In this section, we list the requirements and specifications of the gas burner system.

B.1.1 Construction of the goal-model

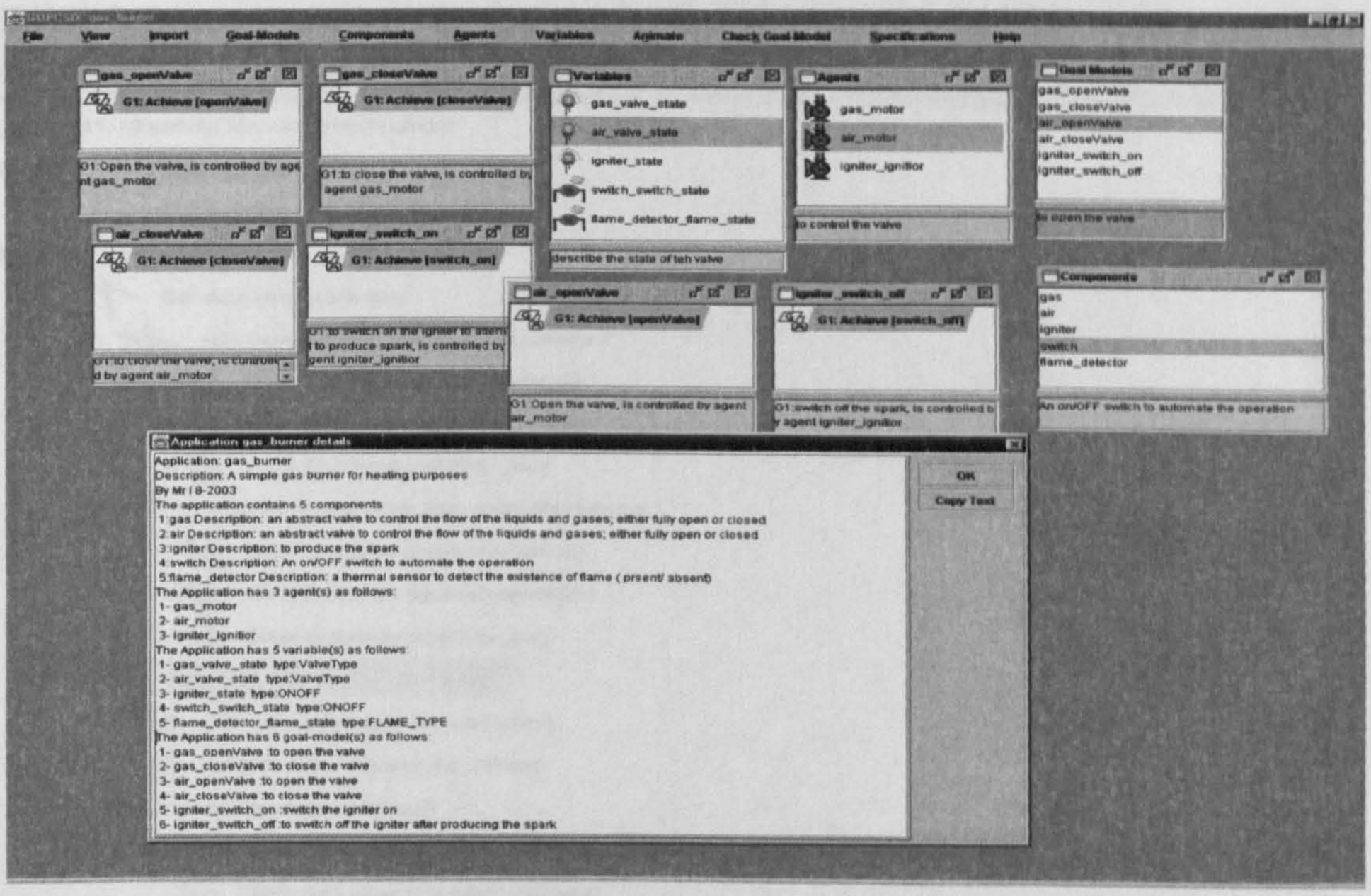


Figure B.1, the desktop of the GOPCSD tool after importing the gas burner components

General descriptuion of the gas burnr system after importing the cmponents

Application: gas_burner
Description: A simple gas burner for heating purposes
By Mr I El-Maddah, 1-8-2003

The application contains 5 components
1-gas Description: an abstract valve to control the flow of the liquids and gases
2-air Description: an abstract valve to control the flow of the liquids and gases
3-igniter Description: to produce the spark
4-switch Description: An on/OFF switch to automate the operation
5-flame_detector Description: a thermal sensor to detect the existence of flame (prsent/ absent)

The Application has 3 agent(s) as follows:
1- gas_motor
2- air_motor
3- igniter_ignitior

The Application has 5 variable(s) as follows:
1- gas_valve_state type:ValveType
2- air_valve_state type: ValveType
3- igniter_state type:ONOFF
4- switch_state type:ONOFF
5- flame_detector_state type:FLAME_TYPE

The Application has 6 goal-model(s) as follows:
1- gas_openValve :to open the valve
2- gas_closeValve :to close the valve
3- air_openValve :to open the valve
4- air_closeValve :to close the valve
5- igniter_switch_on :switch the igniter on
6- igniter_switch_off :to switch off the igniter after producing the spark

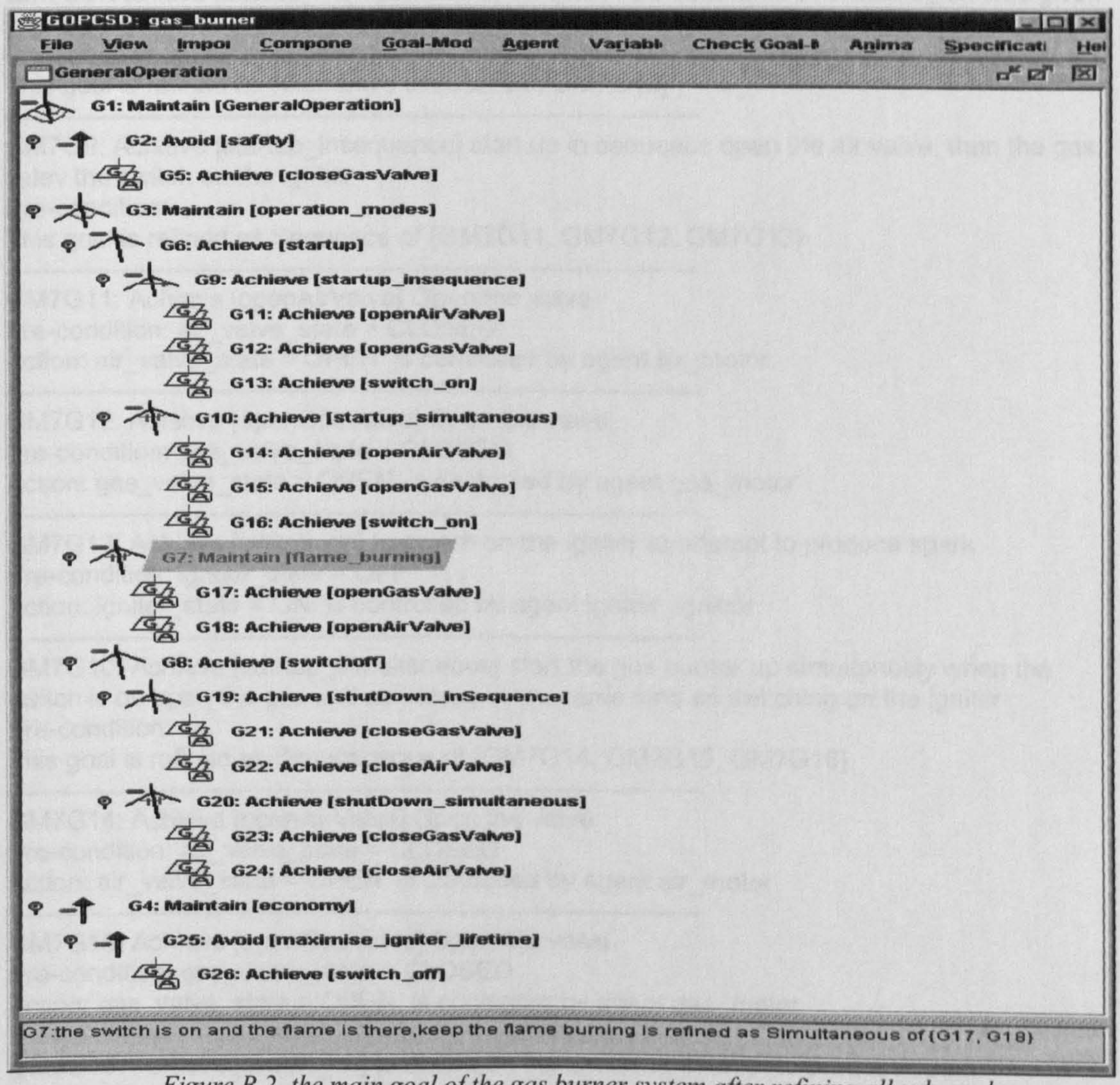


Figure B.2, the main goal of the gas burner system after refining all sub-goal

List of the main goal-model before splitting

Application: gas_burnerAll Goals List:

7- GeneralOperation: describe geenrally the high level functions of the burner

GM7G1: Maintain [GeneralOperation] general functions
Pre-condition:
This goal is refined as Conjunction of {GM7G2, GM7G3, GM7G4}

GM7G2: Avoid [safety] avoid the gas valve open when the air valve is closed
Pre-condition: air_valve_state=CLOSED and gas_valve_state= OPEN
This goal is refined as Inheritance of {GM7G5}

GM7G5: Achieve [closeGasValve] if the gas valve is open and air valve is closed close gas valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM7G3: Maintain [operation_modes]
Pre-condition:
This goal is refined as Disjunction of {GM7G6, GM7G7, GM7G8}

GM7G6: Achieve [startup] the system was off when the command of switching on was given start up
Pre-condition: (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON)
This goal is refined as Alternative of {GM7G9, GM7G10}

GM7G9: Achieve [startup_insequence] start up in sequence open the air valve, then the gas valev the switch on the igniter
Pre-condition:
This goal is refined as Sequence of {GM7G11, GM7G12, GM7G13}

GM7G11: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM7G12: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM7G13: Achieve [switch_on] to switch on the igniter to attempt to produce spark
Pre-condition: igniter_state = OFF
Action: igniter_state = ON is controlled by agent igniter_ignitior

---

GM7G10: Achieve [startup_simultaneous] start the gas burner up simultanosly when the switch is on open the gas and air valves in the same time as switching on the igniter
Pre-condition:
This goal is refined as Simultaneous of {GM7G14, GM7G15, GM7G16}

GM7G14: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM7G15: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM7G16: Achieve [switch_on] to switch on the igniter to attempt to produce spark
Pre-condition: igniter_state = OFF
Action: igniter_state = ON is controlled by agent igniter_ignitior

---

GM7G7: Maintain [flame_burning] the switch is on and the flame is there keep the flame burning
Pre-condition: (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT)
This goal is refined as Simultaneous of {GM7G17, GM7G18}

GM7G17: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM7G18: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM7G8: Achieve [switchoff] the flame was there but the switch is off now switch off
Pre-condition: (switch_switch_state = OFF) and (flame_detector_flame_state = PRESENT)
This goal is refined as Alternative of {GM7G19, GM7G20}

GM7G19: Achieve [shutDown_inSequence] when the switch is off and there is a flame detected shut the gas and the air valves
Pre-condition:
This goal is refined as Sequence of {GM7G21, GM7G22}

GM7G21: Achieve [closeGasValve] to close the valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM7G22: Achieve [closeAirValve] to close the valve
Pre-condition: air_valve_state = OPEN
Action: air_valve_state = CLOSED is controlled by agent air_motor

---

GM7G20: Achieve [shutDown_simultaneous] when there is a flame and the switch is off shut down gas and air valves
Pre-condition:
This goal is refined as Simultaneous of {GM7G23, GM7G24}

GM7G23: Achieve [closeGasValve] to close the valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM7G24: Achieve [closeAirValve] to close the valve
Pre-condition: air_valve_state = OPEN
Action: air_valve_state = CLOSED is controlled by agent air_motor

---

GM7G4: Maintain [economy] if the flame exists and switch is ON economical goal
Pre-condition: flame_detector_flame_state=PRESENT and switch_switch_state= ON
This goal is refined as Inheritance of {GM7G25}

GM7G25: Avoid [maximise_igniter_lifetime] if there is a flame present and the switch is on switch off the spark ignitior
Pre-condition:
This goal is refined as Inheritance of {GM7G26}

GM7G26: Achieve [switch_off] switch off the spark switch off the spark ignitor
Pre-condition: igniter_state = ON
Action: igniter_state = OFF is controlled by agent igniter_ignitior

---

B.1.2 Checking the Requirements

List of the incompleteness cases for the sequential version

Completeness check:

Case no 1: Under the following combination :

air_valve_state = OPEN
igniter_state = ON
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 2: Under the following combination :
air_valve_state = OPEN
igniter_state = OFF
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 3: Under the following combination :
air_valve_state = CLOSED
igniter_state = ON
switch_switch_state = ON
flame_detector_flame_state = PRESENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G14, G4, G18, *G19 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G13, *G14, G4, G18, *G19 assigned it the following value "OPEN"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 4: Under the following combination :
air_valve_state = CLOSED
igniter_state = OFF
switch_switch_state = ON
flame_detector_flame_state = PRESENT
The output variable gas_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G2, *G5, G3, G7, *G14, G4 assigned it the following value "CLOSED"
initially: "CLOSED" The active goal list:, G1, G3, G7, *G13, *G14, G4 assigned it the following value "OPEN"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 5: Under the following combination :
gas_valve_state = OPEN
igniter_state = ON
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 6: Under the following combination :

gas_valve_state = CLOSED
igniter_state = ON
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 7: Under the following combination :
gas_valve_state = OPEN
igniter_state = OFF
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 8: Under the following combination :
gas_valve_state = CLOSED
igniter_state = OFF
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G3 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 9: Under the following combination :
gas_valve_state = OPEN
igniter_state = ON
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3, G8, G15, *G16 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3, G8, G15, *G16 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 10: Under the following combination :
gas_valve_state = OPEN
igniter_state = OFF
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable air_valve_state is not fully determined it is assigned different values as follows
initially: "OPEN" The active goal list:, G1, G3, G8, G15, *G16 assigned it the following value "OPEN"
initially: "CLOSED" The active goal list:, G1, G2, *G5, G3, G8, G15, *G16 assigned it the following value "CLOSED"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 11: Under the following combination :

gas_valve_state = CLOSED
air_valve_state = OPEN
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G6, G9, *G11 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G6, G9, *G11 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 12: Under the following combination :
gas_valve_state = OPEN
air_valve_state = CLOSED
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G2, *G5, G3, G6, G9, *G10 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G2, *G5, G3, G6, G9, *G10 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 13: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = CLOSED
switch_switch_state = ON
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G6, G9, *G10 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G6, G9, *G10 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 14: Under the following combination :
gas_valve_state = OPEN
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 15: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 16: Under the following combination :
gas_valve_state = OPEN
air_valve_state = CLOSED
switch_switch_state = OFF

flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G2, *G5, G3 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G2, *G5, G3 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 17: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = CLOSED
switch_switch_state = OFF
flame_detector_flame_state = ABSENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 18: Under the following combination :
gas_valve_state = OPEN
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G15, *G16 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G8, G15, *G16 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 19: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = OPEN
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G15, *G17 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G3, G8, G15, *G17 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 20: Under the following combination :
gas_valve_state = OPEN
air_valve_state = CLOSED
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G2, *G5, G3, G8, G15, *G16 assigned it the following value "ON"
initially: "OFF" The active goal list:, G1, G2, *G5, G3, G8, G15, *G16 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

Case no 21: Under the following combination :
gas_valve_state = CLOSED
air_valve_state = CLOSED
switch_switch_state = OFF
flame_detector_flame_state = PRESENT
The output variable igniter_state is not fully determined it is assigned different values as follows
initially: "ON" The active goal list:, G1, G3, G8, G15 assigned it the following value "ON"

initially: "OFF" The active goal list:, G1, G3, G8, G15 assigned it the following value "OFF"
A high-level goal should be added as the parent of the current main goal(G1) which should have a functional [grand]child-goal to cover this combination

---

21 cases of missing full output variable determinism have been reported!

The goals of the sequential version

Dependency between the application's variables
Variable gas_valve_state is controlled by the following variables:
air_valve_state
switch_switch_state
flame_detector_flame_state
Variable air_valve_state is controlled by the following variables:
gas_valve_state
switch_switch_state
flame_detector_flame_state
Variable igniter_state is controlled by the following variables:
gas_valve_state
air_valve_state
switch_switch_state
flame_detector_flame_state

Application: gas_burnerAll Goals List:

8- GeneralOperation(split no1 /4) :
GM8G1: Maintain [GeneralOperation] general functions
Pre-condition:
This goal is refined as Conjunction of {GM8G2, GM8G3, GM8G4}

GM8G2: Avoid [safety] avoid the gas valve open when the air valve is closed
Pre-condition: air_valve_state=CLOSED and gas_valve_state= OPEN
This goal is refined as Inheritance of {GM8G5}

GM8G5: Achieve [closeGasValve] if the gas valve is open and air valve is closed close gas valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM8G3: Maintain [operation_modes]
Pre-condition:
This goal is refined as Disjunction of {GM8G6, GM8G7, GM8G8, GM8G9}

GM8G6: Achieve [startup] the system was off when the command of switching on was given start up
Pre-condition: (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON)
This goal is refined as Inheritance of {GM8G10}

GM8G10: Achieve [startup_insequence] start up in sequence open the air valve, then the gas valev the switch on the igniter
Pre-condition:
This goal is refined as Sequence of {GM8G11, GM8G12, GM8G13}

GM8G11: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM8G12: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM8G13: Achieve [switch_on] to switch on the igniter to attempt to produce spark
Pre-condition: igniter_state = OFF
Action: igniter_state = ON is controlled by agent igniter_ignitior

---

GM8G7: Maintain [flame_burning] the switch is on and the flame is there keep the flame burning
Pre-condition: (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT)
This goal is refined as Simultaneous of {GM8G14, GM8G15}

GM8G14: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED and air_valve_state=OPEN
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM8G15: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM8G8: Achieve [switchoff] the flame was there but the switch is off now switch off
Pre-condition: (switch_switch_state = OFF) and (flame_detector_flame_state = PRESENT)
This goal is refined as Inheritance of {GM8G16}

GM8G16: Achieve [shutDown_inSequence] when the switch is off and there is a flame detected shut the gas and the air valves
Pre-condition:
This goal is refined as Sequence of {GM8G17, GM8G18}

GM8G17: Achieve [closeGasValve] to close the valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM8G18: Achieve [closeAirValve] to close the valve
Pre-condition: air_valve_state = OPEN
Action: air_valve_state = CLOSED is controlled by agent air_motor

---

GM8G9: Maintain [ensure_closing] when the burner is shutted down ensure valves and igniter are closed
Pre-condition: switch_switch_state=OFF and flame_detector_flame_state=ABSENT
This goal is refined as Simultaneous of {GM8G19, GM8G20, GM8G21}

GM8G19: Achieve [closeAirValve] to close the valve
Pre-condition: air_valve_state = OPEN
Action: air_valve_state = CLOSED is controlled by agent air_motor

---

GM8G20: Achieve [closeGasValve] to close the valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM8G21: Achieve [switch_off] switch off the spark switch off the spark ignitor
Pre-condition: igniter_state = ON
Action: igniter_state = OFF is controlled by agent igniter_ignitior

---

GM8G4: Maintain [economy] if the flame exists and switch is ON economical goal
Pre-condition: flame_detector_flame_state=PRESENT or (flame_detector_flame_state=ABSENT and (gas_valve_state=CLOSED or air_valve_state=CLOSED))

This goal is refined as Inheritance of {GM8G22}

GM8G22: Avoid [maximise_igniter_lifetime] if there is a flame present and the switch is on switch off the spark ignitior
Pre-condition:
This goal is refined as Inheritance of {GM8G23}

GM8G23: Achieve [switch_off] switch off the spark switch off the spark ignitor
Pre-condition: igniter_state = ON
Action: igniter_state = OFF is controlled by agent igniter_ignitior

The simultaneous version

Completeness check for goal-model 9 GeneralOperation(split no4 /4)

Dependency between the application's variables
Variable gas_valve_state is controlled by the following variables:
air_valve_state
switch_switch_state
flame_detector_flame_state
Variable air_valve_state is controlled by the following variables:
switch_switch_state
flame_detector_flame_state
Variable igniter_state is controlled by the following variables:
switch_switch_state
flame_detector_flame_state

Application: gas_burnerAll Goals List:

9- GeneralOperation(split no4 /4) :
GM9G1: Maintain [GeneralOperation] general functions
Pre-condition:
This goal is refined as Conjunction of {GM9G2, GM9G3, GM9G4}

GM9G2: Avoid [safety] avoid the gas valve open when the air valve is closed
Pre-condition: air_valve_state=CLOSED and gas_valve_state= OPEN
This goal is refined as Inheritance of {GM9G5}

GM9G5: Achieve [closeGasValve] if the gas valve is open and air valve is closed close gas valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM9G3: Maintain [operation_modes]
Pre-condition:
This goal is refined as Disjunction of {GM9G6, GM9G7, GM9G8}

GM9G6: Achieve [startup] the system was off when the command of switching on was given start up
Pre-condition: (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON)
This goal is refined as Inheritance of {GM9G9}

GM9G9: Achieve [startup_simultaneous] start the gas burner up simultanosly when the switch is on open the gas and air valves in the same time as switching on the igniter
Pre-condition:
This goal is refined as Simultaneous of {GM9G10, GM9G11, GM9G12}

GM9G10: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM9G11: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED and air_valve_state=OPEN
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM9G12: Achieve [switch_on] to switch on the igniter to attempt to produce spark
Pre-condition: igniter_state = OFF
Action: igniter_state = ON is controlled by agent igniter_ignitior

---

GM9G7: Maintain [flame_burning] the switch is on and the flame is there keep the flame burning
Pre-condition: (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT)
This goal is refined as Simultaneous of {GM9G13, GM9G14}

GM9G13: Achieve [openGasValve] Open the valve
Pre-condition: gas_valve_state = CLOSED and air_valve_state=OPEN
Action: gas_valve_state = OPEN is controlled by agent gas_motor

---

GM9G14: Achieve [openAirValve] Open the valve
Pre-condition: air_valve_state = CLOSED
Action: air_valve_state = OPEN is controlled by agent air_motor

---

GM9G8: Maintain [ensure_closing] when the burner is shutted down ensure valves and igniter are closed
Pre-condition: switch_switch_state=OFF
This goal is refined as Simultaneous of {GM9G15, GM9G16, GM9G17}

GM9G15: Achieve [closeAirValve] to close the valve
Pre-condition: air_valve_state = OPEN
Action: air_valve_state = CLOSED is controlled by agent air_motor

---

GM9G16: Achieve [closeGasValve] to close the valve
Pre-condition: gas_valve_state = OPEN
Action: gas_valve_state = CLOSED is controlled by agent gas_motor

---

GM9G17: Achieve [switch_off] switch off the spark switch off the spark ignitor
Pre-condition: igniter_state = ON
Action: igniter_state = OFF is controlled by agent igniter_ignitior

---

GM9G4: Maintain [economy] if the flame exists and switch is ON economical goal
Pre-condition: flame_detector_flame_state=PRESENT and switch_switch_state= ON
This goal is refined as Inheritance of {GM9G18}

GM9G18: Avoid [maximise_igniter_lifetime] if there is a flame present and the switch is on switch off the spark ignitior
Pre-condition:
This goal is refined as Inheritance of {GM9G19}

GM9G19: Achieve [switch_off] switch off the spark switch off the spark ignitor
Pre-condition: igniter_state = ON
Action: igniter_state = OFF is controlled by agent igniter_ignitior

B.1.3 The formal Specifications

B.1.3.1 The general operation

B.1.3.1.1 The Simultaneous version

Invariant no. 1 closeGasValve Actor(Agent): gas_motor
/* G5: closeGasValve if the gas valve is open and air valve is closed close gas valve*/
Pre-condition: if the gas valve is open and air valve is closed
(gas_valve_state = OPEN) and air_valve_state=CLOSED and gas_valve_state= OPEN
Post-condition: close gas valve
gas_valve_state = CLOSED

Invariant no. 2 openAirValve Actor(Agent): air_motor
/* G10: openAirValve Open the valve */
Pre-condition: Open the valve
(air_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (
switch_switch_state = ON)

Post-condition:
air_valve_state = OPEN

Invariant no. 3 openGasValve Actor(Agent): gas_motor
/* G11: openGasValve Open the valve */
Pre-condition: Open the valve
(gas_valve_state = CLOSED and air_valve_state=OPEN) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON)
Post-condition:
gas_valve_state = OPEN

Invariant no. 4 switch_on Actor(Agent): igniter_ignitior
/* G12: switch_on to switch on the igniter to attempt to produce spark */
Pre-condition: to switch on the igniter to attempt to produce spark
(igniter_state = OFF) and (flame_detector_flame_state = ABSENT) and (switch_switch_state = ON)
Post-condition:
igniter_state = ON

Invariant no. 5 openGasValve Actor(Agent): gas_motor
/* G13: openGasValve Open the valve */
Pre-condition: Open the valve
(gas_valve_state = CLOSED and air_valve_state=OPEN) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT)
Post-condition:
gas_valve_state = OPEN

Invariant no. 6 openAirValve Actor(Agent): air_motor
/* G14: openAirValve Open the valve */
Pre-condition: Open the valve
(air_valve_state = CLOSED) and (switch_switch_state = ON) and (flame_detector_flame_state = PRESENT)
Post-condition:
air_valve_state = OPEN

Invariant no. 7 closeAirValve Actor(Agent): air_motor
/* G15: closeAirValve to close the valve */
Pre-condition: to close the valve
(air_valve_state = OPEN) and switch_switch_state=OFF
Post-condition:
air_valve_state = CLOSED

Invariant no. 8 closeGasValve Actor(Agent): gas_motor
/* G16: closeGasValve to close the valve */
Pre-condition: to close the valve
(gas_valve_state = OPEN) and switch_switch_state=OFF
Post-condition:
gas_valve_state = CLOSED

Invariant no. 9 switch_off Actor(Agent): igniter_ignitior
/* G17: switch_off switch off the spark switch off the spark ignitor*/
Pre-condition: switch off the spark
(igniter_state = ON) and switch_switch_state=OFF
Post-condition: switch off the spark ignitor
igniter_state = OFF

Invariant no. 10 switch_off Actor(Agent): igniter_ignitior
/* G19: switch_off switch off the spark switch off the spark ignitor*/
Pre-condition: switch off the spark
(igniter_state = ON) and flame_detector_flame_state=PRESENT and switch_switch_state= ON

Post-condition: switch off the spark ignitor
igniter_state = OFF

B.1.3.1.2 The Sequence Version

The operations of goal-model 8 the sequential version

Invariant no. 1 closeGasValve Actor(Agent): gas_motor
/* G5: closeGasValve if the gas valve is open and air valve is closed close gas valve*/
Pre-condition: if the gas valve is open and air valve is closed
(gas_valve_state = OPEN) and air_valve_state=CLOSED and gas_valve_state= OPEN
Post-condition: close gas valve
gas_valve_state = CLOSED

Invariant no. 2 openAirValve Actor(Agent): air_motor
/* G11: openAirValve Open the valve */
Pre-condition: Open the valve
(air_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (
switch_switch_state = ON)
Post-condition:
air_valve_state = OPEN

Invariant no. 3 openGasValve Actor(Agent): gas_motor
/* G12: openGasValve Open the valve */
Pre-condition: Open the valve
(gas_valve_state = CLOSED) and (flame_detector_flame_state = ABSENT) and (
switch_switch_state = ON) and air_valve_state = OPEN
Post-condition:
gas_valve_state = OPEN

Invariant no. 4 switch_on Actor(Agent): igniter_ignitior
/* G13: switch_on to switch on the igniter to attempt to produce spark */
Pre-condition: to switch on the igniter to attempt to produce spark
(igniter_state = OFF) and (flame_detector_flame_state = ABSENT) and (
switch_switch_state = ON) and air_valve_state = OPEN and gas_valve_state = OPEN
Post-condition:
igniter_state = ON

Invariant no. 5 openGasValve Actor(Agent): gas_motor
/* G14: openGasValve Open the valve */
Pre-condition: Open the valve
(gas_valve_state = CLOSED and air_valve_state=OPEN) and (switch_switch_state = ON)
and (flame_detector_flame_state = PRESENT)
Post-condition:
gas_valve_state = OPEN

Invariant no. 6 openAirValve Actor(Agent): air_motor
/* G15: openAirValve Open the valve */
Pre-condition: Open the valve
(air_valve_state = CLOSED) and (switch_switch_state = ON) and (
flame_detector_flame_state = PRESENT)
Post-condition:
air_valve_state = OPEN

Invariant no. 7 closeGasValve Actor(Agent): gas_motor
/* G17: closeGasValve to close the valve */
Pre-condition: to close the valve
(gas_valve_state = OPEN) and (switch_switch_state = OFF) and (
flame_detector_flame_state = PRESENT)
Post-condition:
gas_valve_state = CLOSED

Invariant no. 8 closeAirValve Actor(Agent): air_motor
/* G18: closeAirValve to close the valve */
Pre-condition: to close the valve
(air_valve_state = OPEN) and (switch_switch_state = OFF) and (
flame_detector_flame_state = PRESENT) and gas_valve_state = CLOSED
Post-condition:
air_valve_state = CLOSED

Invariant no. 9 closeAirValve Actor(Agent): air_motor
/* G19: closeAirValve to close the valve */
Pre-condition: to close the valve
(air_valve_state = OPEN) and switch_switch_state=OFF and
flame_detector_flame_state=ABSENT
Post-condition:
air_valve_state = CLOSED

Invariant no. 10 closeGasValve Actor(Agent): gas_motor
/* G20: closeGasValve to close the valve */
Pre-condition: to close the valve
(gas_valve_state = OPEN) and switch_switch_state=OFF and
flame_detector_flame_state=ABSENT
Post-condition:
gas_valve_state = CLOSED

Invariant no. 11 switch_off Actor(Agent): igniter_ignitior
/* G21: switch_off switch off the spark switch off the spark ignitor*/
Pre-condition: switch off the spark
(igniter_state = ON) and switch_switch_state=OFF and
flame_detector_flame_state=ABSENT
Post-condition: switch off the spark ignitor
igniter_state = OFF

Invariant no. 12 switch_off Actor(Agent): igniter_ignitior
/* G23: switch_off switch off the spark switch off the spark ignitor*/
Pre-condition: switch off the spark
(igniter_state = ON) and flame_detector_flame_state=PRESENT or
(flame_detector_flame_state=ABSENT and (gas_valve_state=CLOSED or
air_valve_state=CLOSED))
Post-condition: switch off the spark ignitor
igniter_state = OFF

B.3.1.2 The generated B machines

The simultaneous version

The generated B machines are as follows
1- datatypes machine
2- gas valve actuator machine
3- air valve actuator machine
4- igniter actuator machine
5- maincontroller machine

```
MACHINE  datatypes
SETS
ValveType := {CLOSED, OPEN }  /* to describe on/off valves */;
ONOFF := {OFF, ON }      /* two values for valves */;
FLAME_TYPE := {ABSENT, PRESENT }    /*  */
END
```

```
MACHINE   gas_valve_state_actuator
SEES datatypes
VARIABLES
      gas_valve_state              /* output variable describe the
state of the gas valve */
INVARIANT
      gas_valve_state : ValveType
INITIALISATION
      gas_valve_state := CLOSED
OPERATIONS
set_gas_valve_state_CLOSED =
PRE  gas_valve_state /= CLOSED
THEN
gas_valve_state := CLOSED
END;
set_gas_valve_state_OPEN =
PRE  gas_valve_state /= OPEN
THEN
gas_valve_state := OPEN
END
END
```

```
MACHINE   air_valve_state_actuator
SEES datatypes
VARIABLES
      air_valve_state              /* output variable describe the
state of the air valve */
INVARIANT
      air_valve_state : ValveType
INITIALISATION
      air_valve_state := CLOSED
OPERATIONS
set_air_valve_state_CLOSED =
PRE  air_valve_state /= CLOSED
THEN
air_valve_state := CLOSED
END;
set_air_valve_state_OPEN =
PRE  air_valve_state /= OPEN
THEN
air_valve_state := OPEN
END
END
```

```
MACHINE   igniter_state_actuator
SEES datatypes
VARIABLES
      igniter_state                /* output variable the state of the
igniter */
INVARIANT
      igniter_state : ONOFF
INITIALISATION
      igniter_state := OFF
OPERATIONS
set_igniter_state_OFF =
PRE  igniter_state /= OFF
THEN
igniter_state := OFF
```

```
END;
set_igniter_state_ON =
PRE  igniter_state /= ON
THEN
igniter_state := ON
END
END
```

```
MACHINE  maincontroller
SEES datatypes
INCLUDES
gas_valve_state_actuator , air_valve_state_actuator ,
igniter_state_actuator
OPERATIONS
set_actuators(switch_switch_state , flame_detector_flame_state ) =
PRE
switch_switch_state : ONOFF              /* input variable switch */
 & flame_detector_flame_state : FLAME_TYPE /* input variable flame */
THEN
     IF
 /* G5 closeGasValve:  if the gas valve is open and air valve is closed close gas
valve */
 ( gas_valve_state = OPEN ) & air_valve_state = CLOSED &
gas_valve_state = OPEN
THEN
set_gas_valve_state_CLOSED
ELSIF
 /* G11 openGasValve:  Open the valve  */
 ( gas_valve_state = CLOSED & air_valve_state = OPEN ) & (
flame_detector_flame_state = ABSENT ) & ( switch_switch_state = ON )
THEN
set_gas_valve_state_OPEN
ELSIF
 /* G13 openGasValve:  Open the valve  */
 ( gas_valve_state = CLOSED & air_valve_state = OPEN ) & (
switch_switch_state = ON ) & ( flame_detector_flame_state = PRESENT
)
THEN
set_gas_valve_state_OPEN
ELSIF
 /* G16 closeGasValve:  to close the valve  */
 ( gas_valve_state = OPEN ) & switch_switch_state = OFF
THEN
set_gas_valve_state_CLOSED
     ELSE
SKIP
    END
 ||
     IF
 /* G10 openAirValve:  Open the valve  */
 ( air_valve_state = CLOSED ) & ( flame_detector_flame_state =
ABSENT ) & ( switch_switch_state = ON )
THEN
set_air_valve_state_OPEN
ELSIF
 /* G14 openAirValve:  Open the valve  */
 ( air_valve_state = CLOSED ) & ( switch_switch_state = ON ) & (
flame_detector_flame_state = PRESENT )
THEN
set_air_valve_state_OPEN
ELSIF
 /* G15 closeAirValve:  to close the valve  */
```

```
 ( air_valve_state = OPEN ) & switch_switch_state = OFF
THEN
set_air_valve_state_CLOSED
     ELSE
SKIP
    END
 ||
     IF
 /* G12 switch_on:  to switch on the igniter to attempt to produce spark  */
 ( igniter_state = OFF ) & ( flame_detector_flame_state = ABSENT ) &
( switch_switch_state = ON )
THEN
set_igniter_state_ON
ELSIF
 /* G17 switch_off:  switch off the spark switch off the spark ignitor */
 ( igniter_state = ON ) & switch_switch_state = OFF
THEN
set_igniter_state_OFF
ELSIF
 /* G19 switch_off:  switch off the spark switch off the spark ignitor */
 ( igniter_state = ON ) & flame_detector_flame_state = PRESENT &
switch_switch_state = ON
THEN
set_igniter_state_OFF
     ELSE
SKIP
    END
END
END
```

The sequential version

The controller of the sequential version will be the only variant otherwise the other machines are the same as the simulatnous version

```
MACHINE  maincontroller
SEES datatypes
INCLUDES
gas_valve_state_actuator , air_valve_state_actuator ,
igniter_state_actuator
OPERATIONS
set_actuators(switch_switch_state , flame_detector_flame_state ) =
PRE
switch_switch_state : ONOFF              /* input variable  */
 & flame_detector_flame_state : FLAME_TYPE  /* input variable flame */
THEN
    IF
/* G5 closeGasValve: the gas valve is open and air valve is closed, close gas valve */
 ( gas_valve_state = OPEN ) & air_valve_state = CLOSED &
gas_valve_state = OPEN
THEN
set_gas_valve_state_CLOSED
ELSIF
 /* G12 openGasValve:  Open the valve  */
 ( ( gas_valve_state = CLOSED ) & ( flame_detector_flame_state =
ABSENT ) & ( switch_switch_state = ON ) ) & ( air_valve_state = OPEN
)
THEN
set_gas_valve_state_OPEN
ELSIF
 /* G14 openGasValve:  Open the valve  */
```

```
 ( gas_valve_state = CLOSED & air_valve_state = OPEN ) & (
switch_switch_state = ON ) & ( flame_detector_flame_state = PRESENT
)
THEN
set_gas_valve_state_OPEN
ELSIF
 /* G17 closeGasValve:  to close the valve  */
 ( gas_valve_state = OPEN ) & ( switch_switch_state = OFF ) & (
flame_detector_flame_state = PRESENT )
THEN
set_gas_valve_state_CLOSED
ELSIF
 /* G20 closeGasValve:  to close the valve  */
 ( gas_valve_state = OPEN ) & switch_switch_state = OFF &
flame_detector_flame_state = ABSENT
THEN
set_gas_valve_state_CLOSED
     ELSE
SKIP
    END
 ||
     IF
 /* G11 openAirValve:  Open the valve  */
 ( air_valve_state = CLOSED ) & ( flame_detector_flame_state =
ABSENT ) & ( switch_switch_state = ON )
THEN
set_air_valve_state_OPEN
ELSIF
 /* G15 openAirValve:  Open the valve  */
 ( air_valve_state = CLOSED ) & ( switch_switch_state = ON ) & (
flame_detector_flame_state = PRESENT )
THEN
set_air_valve_state_OPEN
ELSIF
 /* G18 closeAirValve:  to close the valve  */
 ( ( air_valve_state = OPEN ) & ( switch_switch_state = OFF ) & (
flame_detector_flame_state = PRESENT ) ) & ( gas_valve_state =
CLOSED )
THEN
set_air_valve_state_CLOSED
ELSIF
 /* G19 closeAirValve:  to close the valve  */
 ( air_valve_state = OPEN ) & switch_switch_state = OFF &
flame_detector_flame_state = ABSENT
THEN
set_air_valve_state_CLOSED
     ELSE
SKIP
    END
 ||
     IF
 /* G13 switch_on:  to switch on the igniter to attempt to produce spark  */
 ( ( igniter_state = OFF ) & ( flame_detector_flame_state = ABSENT )
& ( switch_switch_state = ON ) ) & ( gas_valve_state = OPEN )
THEN
set_igniter_state_ON
ELSIF
 /* G21 switch_off:  switch off the spark switch off the spark ignitor */
 ( igniter_state = ON ) & switch_switch_state = OFF &
flame_detector_flame_state = ABSENT
THEN
set_igniter_state_OFF
ELSIF
```

```
 /* G23 switch_off:  switch off the spark switch off the spark ignitor */
 ( igniter_state = ON ) & flame_detector_flame_state = PRESENT or (
flame_detector_flame_state = ABSENT & ( gas_valve_state = CLOSED or
air_valve_state = CLOSED ) )
THEN
set_igniter_state_OFF
     ELSE
SKIP
    END
END
END
```

B.2 The Production Cell

B.2.1 Constructing the goal-model

Importing the production cell components

```
Application: Application1
Description: Description of Application 1
By I. El-Maddah 1-9-2003
---------------------------------------------------------------
The application contains 5 components
The details of Component1:Feedbelt
-----------------------------------
Description: belt to convey the products at the different levels from one place to another

The Component has 1 agent(s) as follows:
 1- Feedbelt_motor

The Component has 3 variable(s) as follows:
 1- Feedbelt_moving  type:ONOFF
 2- Feedbelt_metal_at_start  type:YesNo
 3- Feedbelt_metal_at_end  type:YesNo

The Component has 2 goal-model(s) as follows:
 1- Feedbelt_stop_moving :to control the motion of the belt by controling the motor
 2- Feedbelt_start_moving :to contiue moving the products over the belt
---------------------------------------------------------------------------
The details of Component2:Deposibelt
-----------------------------------
Description: belt to convey the products at the different levels from one place to another

The Component has 1 agent(s) as follows:
 1- Deposibelt_motor

The Component has 3 variable(s) as follows:
 1- Deposibelt_moving  type:ONOFF
 2- Deposibelt_metal_at_start  type:YesNo
 3- Deposibelt_metal_at_end  type:YesNo

The Component has 2 goal-model(s) as follows:
 1- Deposibelt_stop_moving :to control the motion of the belt by controling the motor
 2- Deposibelt_start_moving :to contiue moving the products over the belt
---------------------------------------------------------------------------
The details of Component3:Robot
-----------------------------------
Description: Robot to pick and deliver the metals
```

The Component has 5 agent(s) as follows:
1- Robot_r_motor
2- Robot_magnet1
3- Robot_magnet2
4- Robot_motor_1
5- Robot_motor_2

The Component has 8 variable(s) as follows:
1- Robot_r_moving type:ROTATION
2- Robot_arm1_magnet type:ONOFF
3- Robot_arm2_magnet type:ONOFF
4- Robot_arm1_moving type:ARM_MOTION
5- Robot_arm2_moving type:ARM_MOTION
6- Robot_angle type:ROBOT_POSITION
7- Robot_arm1_length type:ARM_POSITION
8- Robot_arm2_length type:ARM_POSITION

The Component has 13 goal-model(s) as follows:
1- Robot_rotate_left :
2- Robot_rotate_right :
3- Robot_stop_rotation :
4- Robot_magnet1_on :
5- Robot_magnet1_off :
6- Robot_magnet2_on :
7- Robot_magnet2_off :
8- Robot_extend_arm1 :
9- Robot_stop_arm1 :
10- Robot_retract_arm1 :
11- Robot_extend_arm2 :
12- Robot_stop_arm2 :
13- Robot_retract_arm2 :

The details of Component4:Table

Description: Table to place the metals to be picked by the Robot

The Component has 2 agent(s) as follows:
1- Table_v_motor
2- Table_r_motor

The Component has 5 variable(s) as follows:
1- Table_has_metal type:YesNo
2- Table_angle type:TABLE_R_POSITION
3- Table_level type:TABLE_POSITION
4- Table_v_moving type:VERTICAL_MOTION
5- Table_r_moving type:ROTATION

The Component has 6 goal-model(s) as follows:
1- Table_move_up :raise the table towards the robot level
2- Table_move_down :move the table to the lower level
3- Table_rotate_right :move the tabel towards the robot arm
4- Table_rotate_left :rotate the table towards the feed belt
5- Table_Stop_moving :stop the vertical motion
6- Table_Stop_rotation :Stop the rotation motion

The details of Component5:Press

Description: Press to stamp the blank metals

The Component has 2 agent(s) as follows:

```
 1- Press_motor
 2- Press_Presser

The Component has 4 variable(s) as follows:
 1- Press_v_moving  type:VERTICAL_MOTION
 2- Press_press_state  type:ONOFF
 3- Press_has_metal  type:YesNo
 4- Press_level  type:PRESS_POSITION

The Component has 5 goal-model(s) as follows:
 1- Press_Move_up :
 2- Press_Move_down :
 3- Press_Stop_moving :
 4- Press_Press_metal :
 5- Press_Stop_Pressing :
```

B.2.2 Checking the Goal-model
Animating the production cell

Table B.1, Event list for testing the production cell normal operation

Cycle no.	Variable name	Value
0	Table_has_metal	YES
1	Table_level	UP
3	Table_angle	ROBOT_FACING
4	Robot_arm1_length	EXTENDED
6	Table_has_metal	NO
7	Robot_arm1_length	RETRACTED
9	Robot_angle	ARM1_PRESS
11	Robot_arm1_length	EXTENDED
13	Press_has_metal	YES
14	Robot_arm1_length	RETRACTED
18	Press_level	UP
20	Press_metal_pressed	YES
23	Press_level	DOWN
25	Robot_angle	ARM2_PRESS
27	Robot_arm2_length	EXTENDED
30	Press_has_metal	NO
30	Press_metal_pressed	NO
31	Robot_arm2_length	RETRACTED
32	Press_level	MIDDLE
33	Robot_angle	ARM2_DEPOSITBELT
35	Robot_arm2_length	EXTENDED

List of the animation result using the above event list

```
Animating goal-model 12 main_goal
 Resetting all Variables to the initial values:
--------------------------------------------------------------------
FeedBelt_moving=ON
FeedBelt_metal_at_start=NO
FeedBelt_metal_at_end=NO
DepositBelt_moving=ON
DepositBelt_metal_at_start=NO
DepositBelt_metal_at_end=NO
Press_v_moving=STOP
Press_press_state=OFF
Press_has_metal=NO
Press_level=MIDDLE
```

Press_metal_pressed=NO
Robot_r_moving=STOP
Robot_arm1_magnet=OFF
Robot_arm2_magnet=OFF
Robot_arm1_moving=STOP
Robot_arm2_moving=STOP
Robot_angle=ARM1_TABLE
Robot_arm1_length=RETRACTED
Robot_arm2_length=RETRACTED
Table_has_metal=NO
Table_angle=FEED_BELT_FACING
Table_level=DOWN
Table_v_moving=STOP
Table_r_moving=STOP

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G21, G22,*G23, G5, G41,*G44, G46, G47, G42, G43, G77,*G89,*G78
Action List:
G44(...=>FeedBelt_moving = ON) Agent: FeedBelt_motor : convey metals to table

---

Interruption :Variable Table_has_metal value has been changed to YES
Cycle:0

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G48, G46, G47, G42, G43, G77,*G78
Action List:----------
G48(...=>Table_v_moving = UP) Agent: Table_v_motor :the table has the metal and it is facing the feed belt move the table up to the ROBOT level

---

Interruption :Variable Table_level value has been changed to UP
Cycle:1

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G49, G46, G47, G42, G43, G77,*G78
Action List:----------
G49(...=>Table_v_moving = STOP) Agent: Table_v_motor :the level of the table is up STOP the vertical motion of the table

---

Cycle:2

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G50, G46, G47, G42, G43, G77,*G78
Action List:----------
G50(...=>Table_r_moving = RIGHT) Agent: Table_r_motor : rotate the table towards the ROBOT to pick the metal

---

Interruption :Variable Table_angle value has been changed to ROBOT_FACING
Cycle:3

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G24, G26, G21,*G23, G5, G41, G45,*G51, G46, G47, G42, G43, G77,*G78
Action List:----------
G51(...=>Table_r_moving = STOP) Agent: Table_r_motor :the table was rotating Stop the roattion of the table

---

Interruption :Variable Robot_arm1_length value has been changed to EXTENDED
Cycle:4

Active List: G1, G2, G6,*G10,*G7, G3,*G14, G16, G4, G20, G26, G21,*G23, G5, G41, G45, G46, G47, G42, G43, G77,*G78

Action List:----------
G10(...=>Table_r_moving = STOP) Agent: Table_r_motor : avoid table collides robot
G7(...=>Robot_r_moving = STOP) Agent: Robot_r_motor :when any of the robot arms is not fully retracted avoid that the robot hits others objects by forbiding the robot rotation
---
Cycle:5

Active List: G1, G2, G6,*G10, G3,*G14, G16, G4, G20, G26, G21,*G23, G5, G41, G45, G46, G54,*G57, G47, G42, G43, G77,*G78
Action List:----------
G10(...=>Table_r_moving = STOP) Agent: Table_r_motor : avoid table collides robot
G57(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 :arm1 is fully extended and magnet 1 is OFF pick the metal (switch the magnet ON)

Interruption :Variable Table_has_metal value has been changed to NO
Cycle:6

Active List: G1, G2, G6,*G10, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G54,*G58, G47, G42, G43, G77,*G78
Action List:----------
G10(...=>Table_r_moving = STOP) Agent: Table_r_motor : avoid table collides robot
G17(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 : keep holding the blank metal
G58(...=>Robot_arm1_moving = RETRACT) Agent: Robot_motor_1 :arm1 is fully extended and holding the metal retract arm1 until it is fully retracted
---
Interruption :Variable Robot_arm1_length value has been changed to RETRACTED
Cycle:7

Active List: G1, G2, G6, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G37,*G23, G5, G41, G46, G54,*G59, G47, G60,*G63, G42, G43, G77,*G89,*G78
Action List:----------
G17(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 : keep holding the blank metal
G37(...=>Table_r_moving = LEFT) Agent: Table_r_motor :the table is facing the robot and has no metal rotate table towards the feed belt to collect new blank metal
G59(...=>Robot_arm1_moving = STOP) Agent: Robot_motor_1 : the arm is retracted stop the robot arm1
G63(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate the robot left to the press
G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt
---
Cycle:8

Active List: G1, G2, G6, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G47, G60, G42, G43, G77,*G78
Action List:----------
G17(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 : keep holding the blank metal
---
Interruption :Variable Robot_angle value has been changed to ARM1_PRESS
Cycle:9

Active List: G1, G2, G6, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G47,*G61, G42, G43, G77,*G78
Action List:----------
G17(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 : keep holding the blank metal
G61(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots first arm is facing the press
---
Cycle:10

Active List: G1, G2, G6, G3,*G14, G16,*G17, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G47, G62,*G65, G42, G43, G77,*G89,*G78
Action List:----------
G17(...=>Robot_arm1_magnet = ON) Agent: Robot_magnet1 : keep holding the blank metal
G65(...=>Robot_arm1_moving = EXTEND) Agent: Robot_motor_1 :arm1 has a metal and is fully retracted Extend arm 1
G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt
---
Interruption :Variable Robot_arm1_length value has been changed to EXTENDED
Cycle:11

Active List: G1, G2, G6,*G7, G3,*G14, G16, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G47,*G61, G42, G43, G77,*G78
Action List:----------
G7(...=>Robot_r_moving = STOP) Agent: Robot_r_motor :when any of the robot arms is not fully retracted avoid that the robot hits others objects by forbiding the robot rotation
G61(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots first arm is facing the press
---
Cycle:12

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G21, G22,*G23, G5, G41, G46, G47, G62,*G66, G42, G43, G77,*G78
Action List:----------
G66(...=>Robot_arm1_moving = STOP) Agent: Robot_motor_1 :

Interruption :Variable Press_has_metal value has been changed to YES
Cycle:13

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G62,*G67, G42, G43, G77,*G78
Action List:----------
G67(...=>Robot_arm1_magnet = OFF) Agent: Robot_magnet1 :arm1 is fully extended and magnet 1 is ON drop the metal (switch the magnet OFF)

Interruption :Variable Robot_arm1_length value has been changed to RETRACTED
Cycle:14

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G62, G42,*G70, G43, G77,*G89,*G78
Action List:----------
G70(...=>Press_v_moving = UP) Agent: Press_motor :there should be a blank metal and the robot arm should be away move the blank metal to the upper place to be processed
G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt
---
Cycle:15

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G70, G43, G77,*G78
Action List:----------
G70(...=>Press_v_moving = UP) Agent: Press_motor :there should be a blank metal and the robot arm should be away move the blank metal to the upper place to be processed
---
Cycle:16

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G70, G43, G77,*G78
Action List:----------

G70(...=>Press_v_moving = UP) Agent: Press_motor :there should be a blank metal and the robot arm should be away move the blank metal to the upper place to be processed

---

Cycle:17

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G70, G43, G77,*G78

Action List:----------

G70(...=>Press_v_moving = UP) Agent: Press_motor :there should be a blank metal and the robot arm should be away move the blank metal to the upper place to be processed

---

Interruption :Variable Press_level value has been changed to UP

Cycle:18

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G71, G43, G77,*G78

Action List:----------

G71(...=>Press_v_moving = STOP) Agent: Press_motor :the press tray in the upper level and has a blank metal Stop the motor

---

Cycle:19

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G72, G43, G77,*G78

Action List:----------

G72(...=>Press_press_state = ON) Agent: Press_Presser : Press the metal

---

Interruption :Variable Press_metal_pressed value has been changed to YES

Cycle:20

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G73, G43, G77,*G78

Action List:----------

G73(...=>Press_press_state = OFF) Agent: Press_Presser :if the metal is pressed Stop Pressing the metal

---

Cycle:21

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42,*G74, G43, G77,*G78

Action List:----------

G74(...=>Press_v_moving = DOWN) Agent: Press_motor :the metal is pressed and in the upper position move the stamped metal to the lower position

---

Cycle:22

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G26, G22,*G23, G5, G41, G47, G42, G43, G77,*G78

Action List:----------

Interruption :Variable Press_level value has been changed to DOWN

Cycle:23

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25, G29,*G32, G26, G22,*G23, G5, G41, G47, G42,*G75, G43, G76, G79,*G82, G77,*G78

Action List:----------

G32(...=>Robot_r_moving = RIGHT) Agent: Robot_r_motor : the robot facing the deposit belt rotate robot to the right until reaching the press

G75(...=>Press_v_moving = STOP) Agent: Press_motor :the press tray in the lowerr level and has a blank metal Stop the motor

G82(...=>Robot_r_moving = RIGHT) Agent: Robot_r_motor : rotate the robot to the right until facing the press

---

Cycle:24

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25, G29, G26, G22,*G23, G5, G41, G47, G42, G43, G76, G79, G77,*G89,*G78

Action List:----------

G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt

---

Interruption :Variable Robot_angle value has been changed to ARM2_PRESS

Cycle:25

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25,*G30, G26, G22,*G23, G5, G41, G47, G42, G43, G76, G79,*G80, G77,*G78

Action List:----------

G30(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots second arm is facing the press

G80(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots first arm is facing the table

---

Cycle:26

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25, G26, G22,*G23, G5, G41, G47, G42, G43, G76, G79, G81,*G84, G77,*G89,*G78

Action List:----------

G84(...=>Robot_arm2_moving = EXTEND) Agent: Robot_motor_2 :arm2 is free and retracted Extend arm 2

G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt

---

Interruption :Variable Robot_arm2_length value has been changed to EXTENDED

Cycle:27

Active List: G1, G2, G6,*G7, G3,*G14, G16, G4, G20, G25,*G30, G26, G22,*G23, G5, G41, G47, G42, G43, G76,*G80, G77,*G78

Action List:----------

G7(...=>Robot_r_moving = STOP) Agent: Robot_r_motor :when any of the robot arms is not fully retracted avoid that the robot hits others objects by forbiding the robot rotation

G30(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots second arm is facing the press

G80(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robots first arm is facing the table

---

Cycle:28

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25, G26, G22,*G23, G5, G41, G47, G42, G43, G76, G81,*G85, G77,*G78

Action List:----------

G85(...=>Robot_arm2_moving = STOP) Agent: Robot_motor_2 :

Cycle:29

Active List: G1, G2, G6, G3,*G14, G16, G4, G20, G25, G26, G22,*G23, G5, G41, G47, G42, G43, G76, G81,*G86, G77,*G78

Action List:----------

G86(...=>Robot_arm2_magnet = ON) Agent: Robot_magnet2 :arm2 is fully extended and magnet 2 is OFF pick the metal (switch the magnet ON)

Interruption :Variable Press_has_metal value has been changed to NO
Interruption :Variable Press_metal_pressed value has been changed to NO
Cycle:30

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21, G22,*G23, G5, G41, G47, G42, G43, G77,*G78
Action List:---

Interruption :Variable Robot_arm2_length value has been changed to RETRACTED
Cycle:31

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21,*G35, G22,*G23, G5, G41, G47, G42, G43, G77,*G89,*G78
Action List:---
G35(...=>Press_v_moving = UP) Agent: Press_motor :the press tray in the lower position and it has no metal the motor moves the press tray to the middle position
G89(...=>Robot_r_moving = LEFT) Agent: Robot_r_motor : rotate to the left until arm2 faces the deposit belt
---
Interruption :Variable Press_level value has been changed to MIDDLE
Cycle:32

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21,*G36, G22,*G23, G5, G41, G47, G42, G43, G77,*G78
Action List:---
G36(...=>Press_v_moving = STOP) Agent: Press_motor :stop the motor the press tray reached its middle position
---
Interruption :Variable Robot_angle value has been changed to ARM2_DEPOSITBELT
Cycle:33

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21, G22,*G23, G5, G41, G47, G42, G43, G77,*G90,*G78
Action List:---
G90(...=>Robot_r_moving = STOP) Agent: Robot_r_motor : stop the motor once the robot's second arm is facing the deposit belt
---
Cycle:34

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21, G22,*G23, G5, G41, G47, G42, G43, G77, G91,*G92,*G78
Action List:---
G92(...=>Robot_arm2_moving = EXTEND) Agent: Robot_motor_2 :arm2 has a metal and is fully retracted Extend arm 2

Interruption :Variable Robot_arm2_length value has been changed to EXTENDED
Cycle:35

Active List: G1, G2, G6, G3,*G14, G4, G20, G26, G21, G22,*G23, G5, G41, G47, G42, G43, G77, G91,*G93,*G78
Action List:---
G93(...=>Robot_arm2_moving = STOP) Agent: Robot_motor_2 :

B.2.3 Generating the Formal Specifications
B.2.3.1 Generating the General Operations

Invariant no. 1 table_collides_robot Actor(Agent): Table_r_motor

/* G10: table_collides_robot avoid table collides robot*/ Pre-condition: Table_level = UP and Table_angle = ROBOT_FACING and Robot_arm1_length = EXTENDED and Robot_angle = ARM1_TABLE Post-condition: avoid table collides robot Table_r_moving = STOP
Invariant no. 2 press_collides_arm1 Actor(Agent): Press_motor
/* G11: press_collides_arm1 avoid the press tray hitting the robot first arm when it is in the middle position*/ Pre-condition: Robot_angle = ARM1_PRESS and Robot_arm1_length = EXTENDED and Press_level = MIDDLE and not (Press_v_moving = STOP) Post-condition: avoid the press tray hitting the robot first arm when it is in the middle position Press_v_moving = STOP
Invariant no. 3 press_collides_arm2 Actor(Agent): Press_motor
/* G12: press_collides_arm2 avoid the press tray hits the robot arm2 when it is in the down position*/ Pre-condition: Robot_angle=ARM2_PRESS and Robot_arm2_length = EXTENDED and Press_level = DOWN and not (Press_v_moving = STOP) Post-condition: avoid the press tray hits the robot arm2 when it is in the down position Press_v_moving = STOP
Invariant no. 4 robot_hits_other_objects Actor(Agent): Robot_r_motor
/* G7: robot_hits_other_objects when any of the robot arms is not fully retracted avoid that the robot hits others objects by forbiding the robot rotation*/ Pre-condition: when any of the robot arms is not fully retracted not (Robot_r_moving = STOP) and (Robot_arm1_length = EXTENDED or Robot_arm2_length = EXTENDED) Post-condition: avoid that the robot hits others objects by forbiding the robot rotation Robot_r_moving = STOP
Invariant no. 5 two_metals_on_table Actor(Agent): FeedBelt_motor
/* G8: two_metals_on_table avoid having two metals on the table*/ Pre-condition: Table_level = DOWN and Table_angle = FEED_BELT_FACING and FeedBelt_moving = ON and Table_has_metal = YES and FeedBelt_metal_at_end = YES Post-condition: avoid having two metals on the table FeedBelt_moving = OFF
Invariant no. 6 two_metals_in_press Actor(Agent): Robot_magnet2
/* G9: two_metals_in_press if the press has already metal and arm2 is extended, avoid having two metals in the press*/ Pre-condition: if the press has already metal and arm2 is extended Robot_arm2_magnet = ON and Robot_arm2_length= EXTENDED and Press_has_metal = YES Post-condition: avoid having two metals in the press Robot_arm2_magnet = ON
Invariant no. 7 dropping_metals_from_feedbelt Actor(Agent): FeedBelt_motor
/* G13: dropping_metals_from_feedbelt the table is not ready stop the feed belt*/ Pre-condition: the table is not ready FeedBelt_metal_at_end =YES and FeedBelt_moving = ON and not (Table_angle = FEED_BELT_FACING and Table_level = DOWN) Post-condition: stop the feed belt FeedBelt_moving = OFF
Invariant no. 8 Pushing_metals_out Actor(Agent): FeedBelt_motor
/* G14: Pushing_metals_out when the balnk metal is on the table and the robot arm is extending donot let the robot pushes the metal away; is condiered as an environemental goal*/ Pre-condition: when the balnk metal is on the table and the robot arm is extending Environmental Assumption : donot let the robot pushes the metal away; is condiered as an environemental goal

Invariant no. 9 Picking_unprocessed_metals Actor(Agent): Robot_magnet2
/* G15: Picking_unprocessed_metals robot arm2 is extended and the press has unprocessed metal avoid the robot picking unprocessed metals*/ Pre-condition: robot arm2 is extended and the press has unprocessed metal Robot_arm1_length = EXTENDED and Robot_angle = ARM2_PRESS and Press_metal_pressed = NO and Press_level= DOWN and Robot_arm2_magnet = OFF Post-condition: avoid the robot picking unprocessed metals Robot_arm2_magnet = OFF
Invariant no. 10 magnet1_on Actor(Agent): Robot_magnet1
/* G17: magnet1_on keep holding the blank metal*/ Pre-condition: (Robot_arm1_magnet = ON and not (Robot_arm1_length =EXTENDED and Robot_angle=ARM1_PRESS)) and Robot_arm1_magnet=ON or Robot_arm2_magnet =OFF Post-condition: keep holding the blank metal Robot_arm1_magnet = ON
Invariant no. 11 magnet2_on Actor(Agent): Robot_magnet2
/* G18: magnet2_on keep holding the processed metal*/ Pre-condition: (Robot_arm2_magnet = ON and not (Robot_angle = ARM2_DEPOSITBELT and Robot_arm2_length =EXTENDED)) and Robot_arm1_magnet=ON or Robot_arm2_magnet =OFF Post-condition: keep holding the processed metal Robot_arm2_magnet = ON
Invariant no. 12 Feedbelt_on Actor(Agent): FeedBelt_motor
/* G19: Feedbelt_on keep the feed belt motor on */ Pre-condition: Table_has_metal= NO and Table_level= DOWN and Table_angle=FEED_BELT_FACING and FeedBelt_moving = OFF Post-condition: keep the feed belt motor on FeedBelt_moving = ON
Invariant no. 13 rotate_right Actor(Agent): Robot_r_motor
/* G27: rotate_right not moving to the right and not facing the table Rotate the robot to the right*/ Pre-condition: not moving to the right and not facing the table (not (Robot_r_moving = RIGHT) and not (Robot_angle= ARM1_TABLE)) and Table_has_metal = YES and Press_has_metal = NO and Robot_arm1_magnet = OFF and Robot_arm2_magnet = OFF and Robot_arm1_length= RETRACTED and Robot_arm2_length=RETRACTED Post-condition: Rotate the robot to the right Robot_r_moving = RIGHT
Invariant no. 14 stop_rotation Actor(Agent): Robot_r_motor
/* G28: stop_rotation stop the motor once the robots first arm is facing the table*/ Pre-condition: (not (Robot_r_moving = STOP) and Robot_angle = ARM1_TABLE) and Table_has_metal = YES and Press_has_metal = NO and Robot_arm1_magnet = OFF and Robot_arm2_magnet = OFF and Robot_arm1_length= RETRACTED and Robot_arm2_length=RETRACTED and Robot_r_moving = RIGHT Post-condition: stop the motor once the robots first arm is facing the table Robot_r_moving = STOP
Invariant no. 15 rotate_leftActor(Agent): Robot_r_motor
/* G31: rotate_left rotate the robot left to the press*/ Pre-condition: (not (Robot_r_moving = LEFT) and Robot_angle = ARM1_TABLE) and (Robot_arm1_length= RETRACTED and Robot_arm2_length= RETRACTED and not (Robot_angle = ARM2_PRESS)) and Robot_arm2_magnet = OFF and Press_has_metal = YES and Press_level = DOWN Post-condition: rotate the robot left to the press Robot_r_moving = LEFT
Invariant no. 16 rotate_right Actor(Agent): Robot_r_motor

/* G32: rotate_right the robot facing the deposit belt rotate robot to the right until reaching the press*/ Pre-condition: the robot facing the deposit belt (not (Robot_r_moving = RIGHT) and (Robot_angel = ARM2_DEPOSITBELT or Robot_angle = ARM1_PRESS)) and (Robot_arm1_length= RETRACTED and Robot_arm2_length= RETRACTED and not (Robot_angle = ARM2_PRESS)) and Robot_arm2_magnet = OFF and Press_has_metal = YES and Press_level = DOWN Post-condition: rotate robot to the right until reaching the press Robot_r_moving = RIGHT
Invariant no. 17 stop_rotation Actor(Agent): Robot_r_motor
/* G30: stop_rotation stop the motor once the robots second arm is facing the press*/ Pre-condition: (not (Robot_r_moving = STOP) and Robot_angle = ARM2_PRESS) and Robot_arm2_magnet = OFF and Press_has_metal = YES and Press_level = DOWN and Post-condition: stop the motor once the robots second arm is facing the press Robot_r_moving = STOP
Invariant no. 18 RetractArm1 Actor(Agent): Robot_motor_1
/* G33: RetractArm1 if arm 1 is not retracted Reatrct Arm 1*/ Pre-condition: if arm 1 is not retracted not (Robot_arm1_length = RETRACTED) and not (Robot_arm1_moving = RETRACT) and not (Robot_angle = ARM1_TABLE) and not (Robot_angle = ARM1_PRESS) Post-condition: Reatrct Arm 1 Robot_arm1_moving = RETRACT
Invariant no. 19 RetractArm2 Actor(Agent): Robot_motor_2
/* G34: RetractArm2 if Arm2 has no metals Retract Arm2*/ Pre-condition: if Arm2 has no metals not (Robot_arm2_length = RETRACTED) and not (Robot_arm2_moving = RETRACT) and not (Robot_angle =ARM2_PRESS) and not (Robot_angle= ARM2_DEPOSITBELT) Post-condition: Retract Arm2 Robot_arm2_moving = RETRACT
Invariant no. 20 MovesUp Actor(Agent): Press_motor
/* G35: MovesUp the press tray in the lower position and it has no metal the motor moves the press tray to the middle position*/ Pre-condition: the press tray in the lower position and it has no metal (Press_has_metal = NO and Press_level = DOWN and not (Press_v_moving = UP) and Robot_arm2_length = RETRACTED) and Press_has_metal= NO Post-condition: the motor moves the press tray to the middle position Press_v_moving = UP
Invariant no. 21 Stops Actor(Agent): Press_motor
/* G36: Stops stop the motor the press tray reached its middle position*/ Pre-condition: stop the motor (not (Press_v_moving = STOP) and Press_has_metal = NO and Press_level = MIDDLE) and Press_has_metal= NO and Press_v_moving = UP Post-condition: the press tray reached its middle position Press_v_moving = STOP
Invariant no. 22 rotate_left Actor(Agent): Table_r_motor
/* G37: rotate_left the table is facing the robot and has no metal rotate table towards the feed belt to collect new blank metal */ Pre-condition: the table is facing the robot and has no metal (Table_angle = ROBOT_FACING and Table_level = UP and Table_has_metal = NO and not (Table_r_moving = LEFT) and not (Robot_arm1_length = EXTENDED)) and Table_has_metal = NO Post-condition: rotate table towards the feed belt to collect new blank metal Table_r_moving = LEFT
Invariant no. 23 Stop_rotation Actor(Agent): Table_r_motor
/* G38: Stop_rotation the table was rotating Stop the roattion of the table*/ Pre-condition: the table was rotating

(Table_angle = FEED_BELT_FACING and Table_level = UP and not (Table_r_moving = STOP)) and Table_has_metal = NO and Table_r_moving = LEFT Post-condition: Stop the roattion of the table Table_r_moving = STOP
Invariant no. 24 move_down Actor(Agent): Table_v_motor
/* G39: move_down move the table down to the feedbelt level*/ Pre-condition: (Table_angle = FEED_BELT_FACING and Table_level = UP and Table_has_metal = NO and not (Table_v_moving = DOWN)) and Table_has_metal = NO and Table_r_moving = LEFT and Table_r_moving = STOP Post-condition: move the table down to the feedbelt level Table_v_moving = DOWN
Invariant no. 25 Stop_moving Actor(Agent): Table_v_motor
/* G40: Stop_moving STOP the vertical motion of the table*/ Pre-condition: (Table_level = DOWN and Table_angle = FEED_BELT_FACING and not (Table_v_moving = STOP)) and Table_has_metal = NO and Table_r_moving = LEFT and Table_r_moving = STOP and Table_v_moving = DOWN Post-condition: STOP the vertical motion of the table Table_v_moving = STOP
Invariant no. 26 Depositbelt_on Actor(Agent): FeedBelt_motor
/* G23: Depositbelt_on keep the deposit belt motor on this is an assumption from the environment*/ Pre-condition: Environmental Assumption : keep the deposit belt motor on this is an assumption from the environment
Invariant no. 27 convey_metals_to_Table Actor(Agent): FeedBelt_motor
/* G44: convey_metals_to_Table convey metals to table */ Pre-condition: Table_has_metal =NO and Table_level=DOWN and Table_angle=FEED_BELT_FACING Post-condition: convey metals to table FeedBelt_moving = ON
Invariant no. 28 move_up Actor(Agent): Table_v_motor
/* G48: move_up the table has the metal and it is facing the feed belt move the table up to the ROBOT level*/ Pre-condition: the table has the metal and it is facing the feed belt (Table_angle = FEED_BELT_FACING and Table_level = DOWN and not (Table_v_moving = UP)) and Table_has_metal= YES Post-condition: move the table up to the ROBOT level Table_v_moving = UP
Invariant no. 29 Stop_moving Actor(Agent): Table_v_motor
/* G49: Stop_moving the level of the table is up STOP the vertical motion of the table*/ Pre-condition: the level of the table is up (Table_level = UP and Table_angle = FEED_BELT_FACING and not (Table_v_moving = STOP)) and Table_has_metal= YES and Table_v_moving = UP Post-condition: STOP the vertical motion of the table Table_v_moving = STOP
Invariant no. 30 rotate_right Actor(Agent): Table_r_motor
/* G50: rotate_right rotate the table towards the ROBOT to pick the metal*/ Pre-condition: (Table_angle = FEED_BELT_FACING and Table_level = UP and not (Table_r_moving = RIGHT)) and Table_has_metal= YES and Table_v_moving = UP and Table_v_moving = STOP Post-condition: rotate the table towards the ROBOT to pick the metal Table_r_moving = RIGHT
Invariant no. 31 Stop_rotation Actor(Agent): Table_r_motor
/* G51: Stop_rotation the table was rotating Stop the roattion of the table*/

Pre-condition: the table was rotating (Table_angle = ROBOT_FACING and not (Table_r_moving = STOP)) and Table_has_metal= YES and Table_v_moving = UP and Table_v_moving = STOP and Table_r_moving = RIGHT Post-condition: Stop the roattion of the table Table_r_moving = STOP
Invariant no. 32 move_robot_to_table Actor(Agent): Robot_r_motor
/* G52: move_robot_to_table move the robot so that arm1 faces the table*/ Pre-condition: (Table_level = UP and Table_has_metal = YES and Table_angle = ROBOT_FACING and not (Robot_r_moving = RIGHT) and not (Robot_angle = ARM1_TABLE) and Robot_arm2_length = RETRACTED and Robot_arm1_length = RETRACTED) and Robot_arm2_magnet =OFF and Press_has_metal = NO Post-condition: move the robot so that arm1 faces the table Robot_r_moving = RIGHT
Invariant no. 33 stop_rotation Actor(Agent): Robot_r_motor
/* G53: stop_rotation stop the motor once the robots first arm is facing the table*/ Pre-condition: (not (Robot_r_moving = STOP) and Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING and Table_has_metal = YES and Table_level = UP and Robot_arm1_magnet = OFF and Robot_arm2_magnet= OFF) and Robot_arm2_magnet =OFF and Press_has_metal = NO and Robot_r_moving = RIGHT Post-condition: stop the motor once the robots first arm is facing the table Robot_r_moving = STOP
Invariant no. 34 Extend_arm1 Actor(Agent): Robot_motor_1
/* G55: Extend_arm1 arm1 is free and retracted Extend arm 1*/ Pre-condition: arm1 is free and retracted (Robot_arm1_length = RETRACTED and Robot_arm1_magnet = OFF and not (Robot_arm1_moving = EXTEND)) and (Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING) and Robot_arm2_magnet =OFF and Press_has_metal = NO and Robot_r_moving = RIGHT and Robot_r_moving = STOP Post-condition: Extend arm 1 Robot_arm1_moving = EXTEND
Invariant no. 35 stop_arm1 Actor(Agent): Robot_motor_1
/* G56: stop_arm1 */ Pre-condition: (not (Robot_arm1_moving = STOP) and Robot_arm1_length = EXTENDED and Robot_arm1_magnet= OFF) and (Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING) and Robot_arm2_magnet =OFF and Press_has_metal = NO and Robot_r_moving = RIGHT and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND Post-condition: Robot_arm1_moving = STOP
Invariant no. 36 ARM1_PICKs Actor(Agent): Robot_magnet1
/* G57: ARM1_PICKs arm1 is fully extended and magnet 1 is OFF pick the metal (switch the magnet ON)*/ Pre-condition: arm1 is fully extended and magnet 1 is OFF (Robot_arm1_length = EXTENDED and Robot_arm1_magnet = OFF) and (Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING) and Robot_arm2_magnet =OFF and Press_has_metal = NO and Robot_r_moving = RIGHT and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP Post-condition: pick the metal (switch the magnet ON) Robot_arm1_magnet = ON
Invariant no. 37 ARM1_RETRACTS Actor(Agent): Robot_motor_1
/* G58: ARM1_RETRACTS arm1 is fully extended and holding the metal retract arm1 until it is fully retracted*/ Pre-condition: arm1 is fully extended and holding the metal (Robot_arm1_length = EXTENDED and Robot_arm1_magnet = ON) and (Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING) and Robot_arm2_magnet =OFF and

Press_has_metal = NO and Robot_r_moving = RIGHT and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP and Robot_arm1_magnet = ON Post-condition: retract arm1 until it is fully retracted Robot_arm1_moving = RETRACT
Invariant no. 38 stop_arm1 Actor(Agent): Robot_motor_1
/* G59: stop_arm1 the arm is retracted stop the robot arm1 */ Pre-condition: the arm is retracted (not (Robot_arm1_moving = STOP) and Robot_arm1_length = RETRACTED and Robot_arm1_magnet = ON) and (Robot_angle = ARM1_TABLE and Table_angle = ROBOT_FACING) and Robot_arm2_magnet =OFF and Press_has_metal = NO and Robot_r_moving = RIGHT and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP and Robot_arm1_magnet = ON and Robot_arm1_moving = RETRACT Post-condition: stop the robot arm1 Robot_arm1_moving = STOP
Invariant no. 39 rotate_left Actor(Agent): Robot_r_motor
/* G63: rotate_left rotate the robot left to the press*/ Pre-condition: (not (Robot_r_moving = LEFT) and (Robot_angle = ARM1_TABLE or Robot_angle = ARM2_PRESS)) and Robot_arm1_magnet = ON and not (Robot_angle = ARM1_PRESS) and Robot_arm1_length=RETRACTED and Robot_arm2_length= RETRACTED and Press_has_metal = NO Post-condition: rotate the robot left to the press Robot_r_moving = LEFT
Invariant no. 40 rotate_right Actor(Agent): Robot_r_motor
/* G64: rotate_right rotates towards the press*/ Pre-condition: (Robot_arm2_magnet =OFF and Robot_angle = ARM2_DEPOSITBELT and not (Robot_r_moving = RIGHT)) and Robot_arm1_magnet = ON and not (Robot_angle = ARM1_PRESS) and Robot_arm1_length=RETRACTED and Robot_arm2_length= RETRACTED and Press_has_metal = NO Post-condition: rotates towards the press Robot_r_moving = RIGHT
Invariant no. 41 stop_rotation Actor(Agent): Robot_r_motor
/* G61: stop_rotation stop the motor once the robots first arm is facing the press*/ Pre-condition: not (Robot_r_moving = STOP) and Robot_angle = ARM1_PRESS and Press_has_metal = NO and Press_level = MIDDLE and Robot_arm1_magnet= ON and Robot_arm2_magnet= OFF and Post-condition: stop the motor once the robots first arm is facing the press Robot_r_moving = STOP
Invariant no. 42 Extend_arm1 Actor(Agent): Robot_motor_1
/* G65: Extend_arm1 arm1 has a metal and is fully retracted Extend arm 1*/ Pre-condition: arm1 has a metal and is fully retracted (Robot_arm1_length = RETRACTED and Robot_arm1_magnet = ON and not (Robot_arm1_moving = EXTEND)) and Robot_angle = ARM1_PRESS and and Robot_r_moving = STOP Post-condition: Extend arm 1 Robot_arm1_moving = EXTEND
Invariant no. 43 stop_arm1 Actor(Agent): Robot_motor_1
/* G66: stop_arm1 */ Pre-condition: (not (Robot_arm1_moving = STOP) and Robot_arm1_length = EXTENDED and Robot_arm1_magnet= ON) and Robot_angle = ARM1_PRESS and and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND Post-condition: Robot_arm1_moving = STOP
Invariant no. 44 ARM1_DROPs Actor(Agent): Robot_magnet1

/* G67: ARM1_DROPs arm1 is fully extended and magnet 1 is ON drop the metal (switch the magnet OFF)*/ Pre-condition: arm1 is fully extended and magnet 1 is ON (Robot_arm1_length = EXTENDED and Robot_arm1_magnet = ON) and Robot_angle = ARM1_PRESS and and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP Post-condition: drop the metal (switch the magnet OFF) Robot_arm1_magnet = OFF
Invariant no. 45 ARM1_RETRACTS Actor(Agent): Robot_motor_1
/* G68: ARM1_RETRACTS arm1 is fully extended and holding the metal retract arm1 until it is fully retracted*/ Pre-condition: arm1 is fully extended and holding the metal (Robot_arm1_length = EXTENDED and Robot_arm1_magnet = OFF) and Robot_angle = ARM1_PRESS and and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP and Robot_arm1_magnet = OFF Post-condition: retract arm1 until it is fully retracted Robot_arm1_moving = RETRACT
Invariant no. 46 stop_arm1 Actor(Agent): Robot_motor_1
/* G69: stop_arm1 stop the arm from retracting*/ Pre-condition: (not (Robot_arm1_moving = STOP) and Robot_arm1_length = RETRACTED and Robot_arm1_magnet= OFF) and Robot_angle = ARM1_PRESS and and Robot_r_moving = STOP and Robot_arm1_moving = EXTEND and Robot_arm1_moving = STOP and Robot_arm1_magnet = OFF and Robot_arm1_moving = RETRACT Post-condition: stop the arm from retracting Robot_arm1_moving = STOP
Invariant no. 47 moveBlankMetalUP Actor(Agent): Press_motor
/* G70: moveBlankMetalUP there should be a blank metal and the robot arm should be away move the blank metal to the upper place to be processed*/ Pre-condition: there should be a blank metal and the robot arm should be away Press_has_metal = YES and Press_level = MIDDLE and Press_metal_pressed = NO and Robot_arm1_length = RETRACTED Post-condition: move the blank metal to the upper place to be processed Press_v_moving = UP
Invariant no. 48 stoptray Actor(Agent): Press_motor
/* G71: stoptray the press tray in the upper level and has a blank metal Stop the motor*/ Pre-condition: the press tray in the upper level and has a blank metal Press_level = UP and Press_has_metal = YES and Press_metal_pressed = NO and not (Press_v_moving = STOP) and Press_v_moving = UP
Post-condition: Stop the motor Press_v_moving = STOP
Invariant no. 49 PressMetal Actor(Agent): Press_Presser
/* G72: PressMetal Press the metal*/ Pre-condition: Press_press_state = OFF and Press_has_metal = YES and Press_metal_pressed = NO and Press_level = UP and Press_v_moving = UP and Press_v_moving = STOP Post-condition: Press the metal Press_press_state = ON
Invariant no. 50 stopPressing Actor(Agent): Press_Presser
/* G73: stopPressing if the metal is pressed Stop Pressing the metal*/ Pre-condition: if the metal is pressed Press_press_state = ON and Press_metal_pressed = YES and Press_level = UP and Press_v_moving = UP and Press_v_moving = STOP and Press_press_state = ON Post-condition: Stop Pressing the metal Press_press_state = OFF
Invariant no. 51 movestampedMetalDown Actor(Agent): Press_motor

/* G74: movestampedMetalDown the metal is pressed and in the upper position move the stamped metal to the lower position*/ Pre-condition: the metal is pressed and in the upper position Press_has_metal = YES and not (Press_v_moving = DOWN) and Press_level = UP and Press_metal_pressed = YES and Press_v_moving = UP and Press_v_moving = STOP and Press_press_state = ON and Press_press_state = OFF Post-condition: move the stamped metal to the lower position Press_v_moving = DOWN
Invariant no. 52 stoptray Actor(Agent): Press_motor
/* G75: stoptray the press tray in the lowerr level and has a blank metal Stop the motor*/ Pre-condition: the press tray in the lowerr level and has a blank metal Press_level = DOWN and Press_has_metal = YES and Press_v_moving = UP and Press_v_moving = STOP and Press_press_state = ON and Press_press_state = OFF and Press_v_moving = DOWN Post-condition: Stop the motor Press_v_moving = STOP
Invariant no. 53 rotate_right Actor(Agent): Robot_r_motor
/* G82: rotate_right rotate the robot to the right until facing the press*/ Pre-condition: (not (Robot_r_moving = RIGHT) and (Robot_angle = ARM2_DEPOSITBELT or Robot_angle = ARM1_PRESS)) and (Robot_arm2_magnet= OFF and Press_has_metal = YES and Robot_arm1_length = RETRACTED and Robot_arm2_length= RETRACTED) and Press_has_metal = YES and Press_level = DOWN Post-condition: rotate the robot to the right until facing the press Robot_r_moving = RIGHT
Invariant no. 54 rotate_left Actor(Agent): Robot_r_motor
/* G83: rotate_left rotate the robot to the left until arm2 faces the press*/ Pre-condition: (not (Robot_r_moving = LEFT) and Robot_angle = ARM1_TABLE) and (Robot_arm2_magnet= OFF and Press_has_metal = YES and Robot_arm1_length = RETRACTED and Robot_arm2_length= RETRACTED) and Press_has_metal = YES and Press_level = DOWN Post-condition: rotate the robot to the left until arm2 faces the press Robot_r_moving = LEFT
Invariant no. 55 stop_rotation Actor(Agent): Robot_r_motor
/* G80: stop_rotation stop the motor once the robots first arm is facing the table*/ Pre-condition: (not (Robot_r_moving = STOP) and Robot_angle = ARM2_PRESS and Robot_arm2_magnet= OFF) and Press_has_metal = YES and Press_level = DOWN and Post-condition: stop the motor once the robots first arm is facing the table Robot_r_moving = STOP
Invariant no. 56 Extend_arm2 Actor(Agent): Robot_motor_2
/* G84: Extend_arm2 arm2 is free and retracted Extend arm 2*/ Pre-condition: arm2 is free and retracted (Robot_arm2_length = RETRACTED and Robot_arm2_magnet = OFF and not (Robot_arm2_moving = EXTEND)) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and and Robot_r_moving = STOP Post-condition: Extend arm 2 Robot_arm2_moving = EXTEND
Invariant no. 57 stop_arm2 Actor(Agent): Robot_motor_2
/* G85: stop_arm2 */ Pre-condition: (not (Robot_arm2_moving = STOP) and Robot_arm2_length = EXTENDED and Robot_arm2_magnet= OFF) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND Post-condition: Robot_arm2_moving = STOP

Invariant no. 58 ARM2_PICKs Actor(Agent): Robot_magnet2
/* G86: ARM2_PICKs arm2 is fully extended and magnet 2 is OFF pick the metal (switch the magnet ON)*/ Pre-condition: arm2 is fully extended and magnet 2 is OFF (Robot_arm2_length = EXTENDED and Robot_arm2_magnet = OFF and Press_metal_pressed=YES) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP Post-condition: pick the metal (switch the magnet ON) Robot_arm2_magnet = ON
Invariant no. 59 ARM2_RETRACTS Actor(Agent): Robot_motor_2
/* G87: ARM2_RETRACTS arm2 is fully extended and holding the metal retract arm2 until it is fully retracted*/ Pre-condition: arm2 is fully extended and holding the metal (Robot_arm2_length = EXTENDED and Robot_arm2_magnet = ON) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP and Robot_arm2_magnet = ON Post-condition: retract arm2 until it is fully retracted Robot_arm2_moving = RETRACT
Invariant no. 60 stop_arm2 Actor(Agent): Robot_motor_2
/* G88: stop_arm2 stop the arm 2 from further retracting*/ Pre-condition: (not (Robot_arm2_moving = STOP) and Robot_arm2_length = RETRACTED and Robot_arm2_magnet= ON) and (Robot_angle =ARM2_PRESS and Press_level = DOWN) and Press_has_metal = YES and Press_level = DOWN and and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP and Robot_arm2_magnet = ON and Robot_arm2_moving = RETRACT Post-condition: stop the arm 2 from further retracting Robot_arm2_moving = STOP
Invariant no. 61 rotate_left_to_depositbelt Actor(Agent): Robot_r_motor
/* G89: rotate_left_to_depositbelt rotate to the left until arm2 faces the deposit belt*/ Pre-condition: not (Robot_angle = ARM2_DEPOSITBELT) and not (Robot_r_moving = LEFT) and Robot_arm1_length=RETRACTED and Robot_arm2_length= RETRACTED and Robot_arm2_magnet =ON Post-condition: rotate to the left until arm2 faces the deposit belt Robot_r_moving = LEFT
Invariant no. 62 stop_rotation Actor(Agent): Robot_r_motor
/* G90: stop_rotation stop the motor once the robot's second arm is facing the deposit belt*/ Pre-condition: not (Robot_r_moving = STOP) and Robot_angle = ARM2_DEPOSITBELT and Robot_arm2_magnet = ON and Robot_r_moving = LEFT Post-condition: stop the motor once the robot's second arm is facing the deposit belt Robot_r_moving = STOP
Invariant no. 63 Extend_arm2 Actor(Agent): Robot_motor_2
/* G92: Extend_arm2 arm2 has a metal and is fully retracted Extend arm 2*/ Pre-condition: arm2 has a metal and is fully retracted (Robot_arm2_length = RETRACTED and Robot_arm2_magnet = ON and not (Robot_arm2_moving = EXTEND)) and Robot_angle = ARM2_DEPOSITBELT and Robot_r_moving = LEFT and Robot_r_moving = STOP Post-condition: Extend arm 2 Robot_arm2_moving = EXTEND ---------- Invariant no. 64 stop_arm2 Actor(Agent): Robot_motor_2 /* G93: stop_arm2 */ Pre-condition:

(not (Robot_arm2_moving = STOP) and Robot_arm2_length = EXTENDED and Robot_arm2_magnet =ON) and Robot_angle = ARM2_DEPOSITBELT and Robot_r_moving = LEFT and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND
Post-condition:
Robot_arm2_moving = STOP

Invariant no. 65 ARM2_DROPS Actor(Agent): Robot_magnet2
/* G94: ARM2_DROPS arm2 is fully extended and magnet 2 is ON pick the metal (switch the magnet ON)*/
Pre-condition: arm2 is fully extended and magnet 2 is ON
(Robot_arm2_length = EXTENDED and Robot_arm2_magnet = ON and Press_has_metal = NO) and Robot_angle = ARM2_DEPOSITBELT and Robot_r_moving = LEFT and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP
Post-condition: pick the metal (switch the magnet ON)
Robot_arm2_magnet = OFF

Invariant no. 66 ARM2_RETRACTS Actor(Agent): Robot_motor_2
/* G95: ARM2_RETRACTS arm2 is fully extended and holding the metal retract arm2 until it is fully retracted*/
Pre-condition: arm2 is fully extended and holding the metal
(Robot_arm2_length = EXTENDED and Robot_arm2_magnet = OFF) and Robot_angle = ARM2_DEPOSITBELT and Robot_r_moving = LEFT and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP and Robot_arm2_magnet = OFF
Post-condition: retract arm2 until it is fully retracted
Robot_arm2_moving = RETRACT

Invariant no. 67 stop_arm2 Actor(Agent): Robot_motor_2

/* G96: stop_arm2 stop the arm from further retracting*/
Pre-condition:
(not (Robot_arm2_moving = STOP) and Robot_arm2_length = RETRACTED and Robot_arm2_magnet= OFF) and Robot_angle = ARM2_DEPOSITBELT and Robot_r_moving = LEFT and Robot_r_moving = STOP and Robot_arm2_moving = EXTEND and Robot_arm2_moving = STOP and Robot_arm2_magnet = OFF and Robot_arm2_moving = RETRACT
Post-condition: stop the arm from further retracting
Robot_arm2_moving = STOP

Invariant no. 68 conveyProcessedMetal Actor(Agent): FeedBelt_motor

/* G78: conveyProcessedMetal always Convey Processed Metals to the colection area by keeping the deposit belt motor switched on.*/
Pre-condition: always
Environmental Assumption : Convey Processed Metals to the colection area by keeping the deposit belt motor switched on.

B.2.3 Generating B machines

12 B machines have been generated
1-datatypes.mch
2-maincontroller.mch
3-FeedBelt_moving_actuator.mch
4-Press_v_moving_actuator.mch
5-Press_press_state_actuator.mch
6-Robot_r_moving_actuator.mch
7-Robot_arm1_magnet_actuator.mch
8-Robot_arm2_magnet_actuator.mch
9-Robot_arm1_moving_actuator.mch
10-Robot_arm2_moving_actuator.mch
11-Table_v_moving_actuator.mch
12-Table_r_moving_actuator.mch

```
MACHINE   datatypes
SETS
ONOFF := {OFF, ON }      /* two values motors */;
VERTICAL_MOTION := {UP, STOP, DOWN }      /* for motors then move up down
or stop */;
PRESS_POSITION := {UP, MIDDLE, DOWN }      /* the positions of the tray
inside the press */;
ROTATION := {RIGHT, STOP, LEFT }     /* three values for rotary motors */;
ARM_MOTION := {EXTEND, STOP, RETRACT }     /* for the robot arms control */;
ROBOT_POSITIONS := {ARM1_TABLE, ARM1_PRESS, ARM2_PRESS,
ARM2_DEPOSITBELT }      /* Robot positions */;
ARM_POSITION := {RETRACTED, EXTENDED }     /* the robot arm positions */;
TABLE_R_POSITION := {FEED_BELT_FACING, ROBOT_FACING } /* the angel of
the table */;
TABLE_POSITION := {UP, DOWN } /* the rotatry table vertical levels down
aligned to the feed belt up aligned to the robot */
END
```

```
MACHINE  maincontroller
SEES datatypes
INCLUDES
FeedBelt_moving_actuator , DepositBelt_moving_actuator ,
Press_v_moving_actuator , Press_press_state_actuator ,
Robot_r_moving_actuator , Robot_arm1_magnet_actuator ,
Robot_arm2_magnet_actuator , Robot_arm1_moving_actuator ,
Robot_arm2_moving_actuator , Table_v_moving_actuator ,
Table_r_moving_actuator
OPERATIONS
set_actuators(FeedBelt_metal_at_end , Press_has_metal , Press_level
, Press_metal_pressed , Robot_angle , Robot_arm1_length ,
Robot_arm2_length , Table_has_metal , Table_angle , Table_level ) =

PRE
FeedBelt_metal_at_end : YesNo  /* input variable photo cell to indicate
whether there is ametal at the end of the belt */
 & Press_has_metal : YesNo     /* input variable to determine whether there is
ametal inside the press or not */
 & Press_level : PRESS_POSITION /* input variable to indicate the press level
*/
 & Press_metal_pressed : YesNo             /* input variable indicate whether
the metal inside the PRess has been pressed or not yet */
 & Robot_angle : ROBOT_POSITIONS   /* input variable the robot angel */
 & Robot_arm1_length : ARM_POSITION            /* input variable  */
 & Robot_arm2_length : ARM_POSITION           /* input variable  */
 & Table_has_metal : YesNo       /* output variable to determine whether the
table has metal or not */
 & Table_angle : TABLE_R_POSITION  /* input variable to indicate how far the
tabel rotates */
 & Table_level : TABLE_POSITION                /* input variable to indicate the
tabel level up level like robot or lower level like feed belt */
THEN
      IF
 /* G8 two_metals_on_table:   avoid having two metals on the table */
 Table_level = DOWN & Table_angle = FEED_BELT_FACING &
FeedBelt_moving = ON & Table_has_metal = YES & FeedBelt_metal_at_end
= YES
THEN
set_FeedBelt_moving_OFF
ELSIF
 /* G13 dropping_metals_from_feedbelt:  the table is not ready  stop the feed belt */
 FeedBelt_metal_at_end = YES & FeedBelt_moving = ON & not (
Table_angle = FEED_BELT_FACING & Table_level = DOWN )
```

```
THEN
set_FeedBelt_moving_OFF
ELSIF
 /* G19 Feedbelt_on:   keep the feed belt motor on  */
 Table_has_metal = NO & Table_level = DOWN & Table_angle =
FEED_BELT_FACING & FeedBelt_moving = OFF
THEN
set_FeedBelt_moving_ON
ELSIF
 /* G44 convey_metals_to_Table:    convey metals to table  */
 Table_has_metal = NO & Table_level = DOWN & Table_angle =
FEED_BELT_FACING
THEN
set_FeedBelt_moving_ON
     ELSE
SKIP
    END
 ||
     IF
 /* G11 press_collides_arm1:   avoid the press tray hitting the robot first arm when
it is in the middle position */
 Robot_angle = ARM1_PRESS & Robot_arm1_length = EXTENDED &
Press_level = MIDDLE & not ( Press_v_moving = STOP )
THEN
set_Press_v_moving_STOP
ELSIF
 /* G12 press_collides_arm2:   avoid the press tray hits the robot arm2 when it is in
the down position */
 Robot_angle = ARM2_PRESS & Robot_arm2_length = EXTENDED &
Press_level = DOWN & not ( Press_v_moving = STOP )
THEN
set_Press_v_moving_STOP
ELSIF
 /* G35 MovesUp:  the press tray in the lower position and it has no metal the motor
moves the press tray to the middle position */
 ( Press_has_metal = NO & Press_level = DOWN & not ( Press_v_moving
= UP ) & Robot_arm2_length = RETRACTED ) & Press_has_metal = NO
THEN
set_Press_v_moving_UP
ELSIF
 /* G36 Stops:  stop the motor the press tray reached its middle position */
 (( not( Press_v_moving = STOP ) & Press_has_metal = NO &
Press_level = MIDDLE ) & Press_has_metal = NO ) & ( Press_v_moving =
UP )
THEN
set_Press_v_moving_STOP
ELSIF
 /* G70 moveBlankMetalUP:  there should be a blank metal and the robot arm should be
away move the blank metal to the upper place to be processed */
 Press_has_metal = YES & Press_level = MIDDLE & Press_metal_pressed
= NO & Robot_arm1_length = RETRACTED
THEN
set_Press_v_moving_UP
ELSIF
 /* G71 stoptray:  the press tray in the upper level and has a blank metal Stop the
motor */
 ( Press_level = UP & Press_has_metal = YES & Press_metal_pressed =
NO & not ( Press_v_moving = STOP ) ) & ( Press_v_moving = UP )
THEN
set_Press_v_moving_STOP
ELSIF
 /* G74 movestampedMetalDown:  the metal is pressed and in the upper position move the
stamped metal to the lower position */
```

```
 ( Press_has_metal = YES & not ( Press_v_moving = DOWN ) &
Press_level = UP & Press_metal_pressed = YES ) & ( Press_press_state
= OFF )
THEN
set_Press_v_moving_DOWN
ELSIF
 /* G75 stoptray:  the press tray in the lowerr level and has a blank metal Stop the
motor */
 ( Press_level = DOWN & Press_has_metal = YES ) & ( Press_v_moving =
DOWN )
THEN
set_Press_v_moving_STOP
     ELSE
SKIP
    END
 ||
     IF
 /* G72 PressMetal:   Press the metal */
 ( Press_press_state = OFF & Press_has_metal = YES &
Press_metal_pressed = NO & Press_level = UP ) & ( Press_v_moving =
STOP )
THEN
set_Press_press_state_ON
ELSIF
 /* G73 stopPressing:  if the metal is pressed Stop Pressing the metal */
 ( Press_press_state = ON & Press_metal_pressed = YES & Press_level
= UP ) & ( Press_press_state = ON )
THEN
set_Press_press_state_OFF
     ELSE
SKIP
    END
 ||
     IF
 /* G7 robot_hits_other_objects:  when any of the robot arms is not fully retracted
avoid that the robot hits others objects by forbiding the robot rotation */
 not ( Robot_r_moving = STOP ) & ( Robot_arm1_length = EXTENDED or
Robot_arm2_length = EXTENDED )
THEN
set_Robot_r_moving_STOP
ELSIF
 /* G27 rotate_right:   not moving to the right and not facing the table  Rotate the
robot to the right */
 ( not ( Robot_r_moving = RIGHT ) & not ( Robot_angle = ARM1_TABLE )
) & Table_has_metal = YES & Press_has_metal = NO & Robot_arm1_magnet
= OFF & Robot_arm2_magnet = OFF & Robot_arm1_length = RETRACTED &
Robot_arm2_length = RETRACTED
THEN
set_Robot_r_moving_RIGHT
ELSIF
 /* G28 stop_rotation: stop the motor once the robots first arm is facing the table */
 ( ( not ( Robot_r_moving = STOP ) & Robot_angle = ARM1_TABLE ) &
Table_has_metal = YES & Press_has_metal = NO & Robot_arm1_magnet =
OFF & Robot_arm2_magnet = OFF & Robot_arm1_length = RETRACTED &
Robot_arm2_length = RETRACTED ) & ( Robot_r_moving = RIGHT )
THEN
set_Robot_r_moving_STOP
ELSIF
 /* G31 rotate_left:   rotate the robot left to the press */
 ( not ( Robot_r_moving = LEFT ) & Robot_angle = ARM1_TABLE ) & (
Robot_arm1_length = RETRACTED & Robot_arm2_length = RETRACTED & not
( Robot_angle = ARM2_PRESS ) ) & Robot_arm2_magnet = OFF &
Press_has_metal = YES & Press_level = DOWN
```

```
THEN
set_Robot_r_moving_LEFT
ELSIF
 /* G32 rotate_right: the robot facing the deposit belt  rotate robot to the right
until reaching the press */
 ( not ( Robot_r_moving = RIGHT ) & ( Robot_angel = ARM2_DEPOSITBELT
or Robot_angle = ARM1_PRESS ) ) & ( Robot_arm1_length = RETRACTED &
Robot_arm2_length = RETRACTED & not ( Robot_angle = ARM2_PRESS ) ) &
Robot_arm2_magnet = OFF & Press_has_metal = YES & Press_level = DOWN
THEN
set_Robot_r_moving_RIGHT
ELSIF
 /* G30 stop_rotation:stop the motor once the robots second arm is facing the press */
 ( not ( Robot_r_moving = STOP ) & Robot_angle = ARM2_PRESS ) &
Robot_arm2_magnet = OFF & Press_has_metal = YES & Press_level = DOWN
THEN
set_Robot_r_moving_STOP
ELSIF
 /* G52 move_robot_to_table:   move the robot so that arm1 faces the table */
 ( Table_level = UP & Table_has_metal = YES & Table_angle =
ROBOT_FACING & not ( Robot_r_moving = RIGHT ) & not ( Robot_angle =
ARM1_TABLE ) & Robot_arm2_length = RETRACTED & Robot_arm1_length =
RETRACTED ) & Robot_arm2_magnet = OFF & Press_has_metal = NO
THEN
set_Robot_r_moving_RIGHT
ELSIF
 /* G53 stop_rotation: stop the motor once the robots first arm is facing the table */
 ( ( not ( Robot_r_moving = STOP ) & Robot_angle = ARM1_TABLE &
Table_angle = ROBOT_FACING & Table_has_metal = YES & Table_level =
UP & Robot_arm1_magnet = OFF & Robot_arm2_magnet = OFF ) &
Robot_arm2_magnet = OFF & Press_has_metal = NO ) & ( Robot_r_moving
= RIGHT )
THEN
set_Robot_r_moving_STOP
ELSIF
 /* G63 rotate_left:   rotate the robot left to the press */
 ( not ( Robot_r_moving = LEFT ) & ( Robot_angle = ARM1_TABLE or
Robot_angle = ARM2_PRESS ) ) & Robot_arm1_magnet = ON & not (
Robot_angle = ARM1_PRESS ) & Robot_arm1_length = RETRACTED &
Robot_arm2_length = RETRACTED & Press_has_metal = NO
THEN
set_Robot_r_moving_LEFT
ELSIF
 /* G64 rotate_right:    rotates towards the press */
 ( Robot_arm2_magnet = OFF & Robot_angle = ARM2_DEPOSITBELT & not (
Robot_r_moving = RIGHT ) ) & Robot_arm1_magnet = ON & not (
Robot_angle = ARM1_PRESS ) & Robot_arm1_length = RETRACTED &
Robot_arm2_length = RETRACTED & Press_has_metal = NO
THEN
set_Robot_r_moving_RIGHT
ELSIF
 /* G61 stop_rotation: stop the motor once the robots first arm is facing the press */
 not ( Robot_r_moving = STOP ) & Robot_angle = ARM1_PRESS &
Press_has_metal = NO & Press_level = MIDDLE & Robot_arm1_magnet = ON
& Robot_arm2_magnet = OFF
THEN
set_Robot_r_moving_STOP
ELSIF
 /* G82 rotate_right:rotate the robot to the right until facing the press */
 ( not ( Robot_r_moving = RIGHT ) & ( Robot_angle = ARM2_DEPOSITBELT
or Robot_angle = ARM1_PRESS ) ) & ( Robot_arm2_magnet = OFF &
Press_has_metal = YES & Robot_arm1_length = RETRACTED &
```

```
Robot_arm2_length = RETRACTED ) & Press_has_metal = YES &
Press_level = DOWN
THEN
set_Robot_r_moving_RIGHT
ELSIF
 /* G83 rotate_left:   rotate the robot to the left until arm2 faces the press */
 ( not ( Robot_r_moving = LEFT ) & Robot_angle = ARM1_TABLE ) & (
Robot_arm2_magnet = OFF & Press_has_metal = YES & Robot_arm1_length
= RETRACTED & Robot_arm2_length = RETRACTED ) & Press_has_metal =
YES & Press_level = DOWN
THEN
set_Robot_r_moving_LEFT
ELSIF
 /* G80 stop_rotation:stop the motor once the robots first arm is facing the table */
 ( not ( Robot_r_moving = STOP ) & Robot_angle = ARM2_PRESS &
Robot_arm2_magnet = OFF ) & Press_has_metal = YES & Press_level =
DOWN
THEN
set_Robot_r_moving_STOP
ELSIF
/* G89 rotate_left_to_depositbelt:rotate left until arm2 faces the deposit belt */
 not ( Robot_angle = ARM2_DEPOSITBELT ) & not ( Robot_r_moving =
LEFT ) & Robot_arm1_length = RETRACTED & Robot_arm2_length =
RETRACTED & Robot_arm2_magnet = ON
THEN
set_Robot_r_moving_LEFT
ELSIF
/* G90 stop_rotation:stop the motor once the robot's arm2 is facing the depositbelt */
 ( not ( Robot_r_moving = STOP ) & Robot_angle = ARM2_DEPOSITBELT &
Robot_arm2_magnet = ON ) & ( Robot_r_moving = LEFT )
THEN
set_Robot_r_moving_STOP
     ELSE
SKIP
    END
 ||
     IF
 /* G17 magnet1_on:   keep holding the blank metal */
 ( Robot_arm1_magnet = ON & not ( Robot_arm1_length = EXTENDED &
Robot_angle = ARM1_PRESS ) ) & Robot_arm1_magnet = ON or
Robot_arm2_magnet = OFF
THEN
set_Robot_arm1_magnet_ON
ELSIF
 /* G57 ARM1_PICKs:  arm1 is fully extended and magnet 1 is OFF pick the metal (
switch the magnet ON) */
 ( ( Robot_arm1_length = EXTENDED & Robot_arm1_magnet = OFF ) & ( (
Robot_angle = ARM1_TABLE & Table_angle = ROBOT_FACING ) &
Robot_arm2_magnet = OFF & Press_has_metal = NO ) & ( Robot_r_moving
= STOP ) ) & ( Robot_arm1_moving = STOP )
THEN
set_Robot_arm1_magnet_ON
ELSIF
 /* G67 ARM1_DROPs:  arm1 is fully extended and magnet 1 is ON drop
the metal ( switch the magnet OFF) */
 ( ( Robot_arm1_length = EXTENDED & Robot_arm1_magnet = ON ) & (
Robot_angle = ARM1_PRESS ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm1_moving = STOP )
THEN
set_Robot_arm1_magnet_OFF
     ELSE
SKIP
```

```
    END
 ||
     IF
 /* G9 two_metals_in_press:  if the press has already metal and arm2 is extended
avoid having two metals in the press */
 Robot_arm2_magnet = ON & Robot_arm2_length = EXTENDED &
Press_has_metal = YES
THEN
set_Robot_arm2_magnet_ON
ELSIF
 /* G15 Picking_unprocessed_metals:  robot arm2 is extended and the press has
unprocessed metal  avoid the robot  picking unprocessed metals */
 Robot_arm1_length = EXTENDED & Robot_angle = ARM2_PRESS &
Press_metal_pressed = NO & Press_level = DOWN & Robot_arm2_magnet =
OFF
THEN
set_Robot_arm2_magnet_OFF
ELSIF
 /* G18 magnet2_on:   keep holding the processed metal */
 ( Robot_arm2_magnet = ON & not ( Robot_angle = ARM2_DEPOSITBELT &
Robot_arm2_length = EXTENDED ) ) & Robot_arm1_magnet = ON or
Robot_arm2_magnet = OFF
THEN
set_Robot_arm2_magnet_ON
ELSIF
 /* G86 ARM2_PICKs:  arm2 is fully extended and magnet 2 is OFF pick the metal (
switch the magnet ON) */
 ( ( Robot_arm2_length = EXTENDED & Robot_arm2_magnet = OFF &
Press_metal_pressed = YES ) & ( ( Robot_angle = ARM2_PRESS &
Press_level = DOWN ) & Press_has_metal = YES & Press_level = DOWN )
& ( Robot_r_moving = STOP ) ) & ( Robot_arm2_moving = STOP )
THEN
set_Robot_arm2_magnet_ON
ELSIF
 /* G94 ARM2_DROPS:  arm2 is fully extended and magnet 2 is ON pick the metal ( switch
the magnet ON) */
 ( ( Robot_arm2_length = EXTENDED & Robot_arm2_magnet = ON &
Press_has_metal = NO ) & ( Robot_angle = ARM2_DEPOSITBELT ) & (
Robot_r_moving = STOP ) ) & ( Robot_arm2_moving = STOP )
THEN
set_Robot_arm2_magnet_OFF
     ELSE
SKIP
    END
 ||
     IF
 /* G33 RetractArm1:   if arm 1 is not retracted   Reatrct Arm 1 */
 not ( Robot_arm1_length = RETRACTED ) & not ( Robot_arm1_moving =
RETRACT ) & not ( Robot_angle = ARM1_TABLE ) & not ( Robot_angle =
ARM1_PRESS )
THEN
set_Robot_arm1_moving_RETRACT
ELSIF
 /* G55 Extend_arm1:  arm1 is free and retracted Extend arm 1 */
 ( Robot_arm1_length = RETRACTED & Robot_arm1_magnet = OFF & not (
Robot_arm1_moving = EXTEND ) ) & ( ( Robot_angle = ARM1_TABLE &
Table_angle = ROBOT_FACING ) & Robot_arm2_magnet = OFF &
Press_has_metal = NO ) & ( Robot_r_moving = STOP )
THEN
set_Robot_arm1_moving_EXTEND
ELSIF
 /* G56 stop_arm1:    */
```

```
 ( ( not ( Robot_arm1_moving = STOP ) & Robot_arm1_length = EXTENDED
& Robot_arm1_magnet = OFF ) & ( ( Robot_angle = ARM1_TABLE &
Table_angle = ROBOT_FACING ) & Robot_arm2_magnet = OFF &
Press_has_metal = NO ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm1_moving = EXTEND )
THEN
set_Robot_arm1_moving_STOP
ELSIF
 /* G58 ARM1_RETRACTS:  arm1 is fully extended and holding the metal retract arm1
until it is fully retracted */
 ( ( Robot_arm1_length = EXTENDED & Robot_arm1_magnet = ON ) & ( (
Robot_angle = ARM1_TABLE & Table_angle = ROBOT_FACING ) &
Robot_arm2_magnet = OFF & Press_has_metal = NO ) & ( Robot_r_moving
= STOP ) ) & ( Robot_arm1_magnet = ON )
THEN
set_Robot_arm1_moving_RETRACT
ELSIF
 /* G59 stop_arm1:   the arm is retracted  stop the robot arm1 */
 ( ( not ( Robot_arm1_moving = STOP ) & Robot_arm1_length =
RETRACTED & Robot_arm1_magnet = ON ) & ( ( Robot_angle = ARM1_TABLE
& Table_angle = ROBOT_FACING ) & Robot_arm2_magnet = OFF &
Press_has_metal = NO ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm1_moving = RETRACT )
THEN
set_Robot_arm1_moving_STOP
ELSIF
 /* G65 Extend_arm1:  arm1 has a metal and is fully retracted Extend arm 1 */
 ( Robot_arm1_length = RETRACTED & Robot_arm1_magnet = ON & not (
Robot_arm1_moving = EXTEND ) ) & ( Robot_angle = ARM1_PRESS ) & (
Robot_r_moving = STOP )
THEN
set_Robot_arm1_moving_EXTEND
ELSIF
 /* G66 stop_arm1:    */
 ( ( not ( Robot_arm1_moving = STOP ) & Robot_arm1_length = EXTENDED
& Robot_arm1_magnet = ON ) & ( Robot_angle = ARM1_PRESS ) & (
Robot_r_moving = STOP ) ) & ( Robot_arm1_moving = EXTEND )
THEN
set_Robot_arm1_moving_STOP
ELSIF
 /* G68 ARM1_RETRACTS:  arm1 is fully extended and holding the metal retract arm1
until it is fully retracted */
 ( ( Robot_arm1_length = EXTENDED & Robot_arm1_magnet = OFF ) & (
Robot_angle = ARM1_PRESS ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm1_magnet = OFF )
THEN
set_Robot_arm1_moving_RETRACT
ELSIF
 /* G69 stop_arm1:    stop the arm from retracting */
 ( ( not ( Robot_arm1_moving = STOP ) & Robot_arm1_length =
RETRACTED & Robot_arm1_magnet = OFF ) & ( Robot_angle = ARM1_PRESS )
& ( Robot_r_moving = STOP ) ) & ( Robot_arm1_moving = RETRACT )
THEN
set_Robot_arm1_moving_STOP
     ELSE
SKIP
    END
 ||
     IF
 /* G34 RetractArm2:  if Arm2 has no metals Retract Arm2 */
 not ( Robot_arm2_length = RETRACTED ) & not ( Robot_arm2_moving =
RETRACT ) & not ( Robot_angle = ARM2_PRESS ) & not ( Robot_angle =
ARM2_DEPOSITBELT )
```

```
THEN
set_Robot_arm2_moving_RETRACT
ELSIF
 /* G84 Extend_arm2:  arm2 is free and retracted Extend arm 2 */
 ( Robot_arm2_length = RETRACTED & Robot_arm2_magnet = OFF & not (
Robot_arm2_moving = EXTEND ) ) & ( ( Robot_angle = ARM2_PRESS &
Press_level = DOWN ) & Press_has_metal = YES & Press_level = DOWN )
& ( Robot_r_moving = STOP )
THEN
set_Robot_arm2_moving_EXTEND
ELSIF
 /* G85 stop_arm2:    */
 ( ( not ( Robot_arm2_moving = STOP ) & Robot_arm2_length = EXTENDED
& Robot_arm2_magnet = OFF ) & ( ( Robot_angle = ARM2_PRESS &
Press_level = DOWN ) & Press_has_metal = YES & Press_level = DOWN )
& ( Robot_r_moving = STOP ) ) & ( Robot_arm2_moving = EXTEND )
THEN
set_Robot_arm2_moving_STOP
ELSIF
 /* G87 ARM2_RETRACTS:  arm2 is fully extended and holding the metal retract arm2
until it is fully retracted */
 ( ( Robot_arm2_length = EXTENDED & Robot_arm2_magnet = ON ) & ( (
Robot_angle = ARM2_PRESS & Press_level = DOWN ) & Press_has_metal =
YES & Press_level = DOWN ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm2_magnet = ON )
THEN
set_Robot_arm2_moving_RETRACT
ELSIF
 /* G88 stop_arm2:    stop the arm 2 from further retracting */
 ( ( not ( Robot_arm2_moving = STOP ) & Robot_arm2_length =
RETRACTED & Robot_arm2_magnet = ON ) & ( ( Robot_angle = ARM2_PRESS
& Press_level = DOWN ) & Press_has_metal = YES & Press_level = DOWN
) & ( Robot_r_moving = STOP ) ) & ( Robot_arm2_moving = RETRACT )
THEN
set_Robot_arm2_moving_STOP
ELSIF
 /* G92 Extend_arm2:  arm2 has a metal and is fully retracted Extend arm 2 */
 ( Robot_arm2_length = RETRACTED & Robot_arm2_magnet = ON & not (
Robot_arm2_moving = EXTEND ) ) & ( Robot_angle = ARM2_DEPOSITBELT )
& ( Robot_r_moving = STOP )
THEN
set_Robot_arm2_moving_EXTEND
ELSIF
 /* G93 stop_arm2:    */
 ( ( not ( Robot_arm2_moving = STOP ) & Robot_arm2_length = EXTENDED
& Robot_arm2_magnet = ON ) & ( Robot_angle = ARM2_DEPOSITBELT ) & (
Robot_r_moving = STOP ) ) & ( Robot_arm2_moving = EXTEND )
THEN
set_Robot_arm2_moving_STOP
ELSIF
 /* G95 ARM2_RETRACTS:  arm2 is fully extended and holding the metal retract arm2
until it is fully retracted */
 ( ( Robot_arm2_length = EXTENDED & Robot_arm2_magnet = OFF ) & (
Robot_angle = ARM2_DEPOSITBELT ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm2_magnet = OFF )
THEN
set_Robot_arm2_moving_RETRACT
ELSIF
 /* G96 stop_arm2:    stop the arm from further retracting */
 ( ( not ( Robot_arm2_moving = STOP ) & Robot_arm2_length =
RETRACTED & Robot_arm2_magnet = OFF ) & ( Robot_angle =
ARM2_DEPOSITBELT ) & ( Robot_r_moving = STOP ) ) & (
Robot_arm2_moving = RETRACT )
```

```
THEN
set_Robot_arm2_moving_STOP
     ELSE
SKIP
    END
 ||
     IF
 /* G39 move_down:   move the table down to the feedbelt level */
 ( ( Table_angle = FEED_BELT_FACING & Table_level = UP &
Table_has_metal = NO & not ( Table_v_moving = DOWN ) ) &
Table_has_metal = NO ) & ( Table_r_moving = STOP )
THEN
set_Table_v_moving_DOWN
ELSIF
 /* G40 Stop_moving:   STOP the vertical motion of the table */
 ( ( Table_level = DOWN & Table_angle = FEED_BELT_FACING & not (
Table_v_moving = STOP ) ) & Table_has_metal = NO ) & (
Table_v_moving = DOWN )
THEN
set_Table_v_moving_STOP
ELSIF
 /* G48 move_up:  the table has the metal and it is facing the feed belt move the
table up to the ROBOT level */
 ( Table_angle = FEED_BELT_FACING & Table_level = DOWN & not (
Table_v_moving = UP ) ) & Table_has_metal = YES
THEN
set_Table_v_moving_UP
ELSIF
 /* G49 Stop_moving:  the level of the table is up STOP the vertical motion of the
table */
 ( ( Table_level = UP & Table_angle = FEED_BELT_FACING & not (
Table_v_moving = STOP ) ) & Table_has_metal = YES ) & (
Table_v_moving = UP )
THEN
set_Table_v_moving_STOP
     ELSE
SKIP
    END
 ||
     IF
 /* G10 table_collides_robot:   avoid table collides robot */
 Table_level = UP & Table_angle = ROBOT_FACING & Robot_arm1_length =
EXTENDED & Robot_angle = ARM1_TABLE
THEN
set_Table_r_moving_STOP
ELSIF
 /* G37 rotate_left:  the table is facing the robot and has no metal rotate table
towards the feed belt to collect new blank metal  */
 ( Table_angle = ROBOT_FACING & Table_level = UP & Table_has_metal =
NO & not ( Table_r_moving = LEFT ) & not ( Robot_arm1_length =
EXTENDED ) ) & Table_has_metal = NO
THEN
set_Table_r_moving_LEFT
ELSIF
 /* G38 Stop_rotation:  the table was rotating Stop the roattion of the table */
 ( ( Table_angle = FEED_BELT_FACING & Table_level = UP & not (
Table_r_moving = STOP ) ) & Table_has_metal = NO ) & (
Table_r_moving = LEFT )
THEN
set_Table_r_moving_STOP
ELSIF
 /* G50 rotate_right:   rotate the table towards the ROBOT to pick the metal */
```

```
 ( ( Table_angle = FEED_BELT_FACING & Table_level = UP & not (
Table_r_moving = RIGHT ) ) & Table_has_metal = YES ) & (
Table_v_moving = STOP )
THEN
set_Table_r_moving_RIGHT
ELSIF
 /* G51 Stop_rotation:  the table was rotating Stop the roattion of the table */
 ( ( Table_angle = ROBOT_FACING & not ( Table_r_moving = STOP ) ) &
Table_has_metal = YES ) & ( Table_r_moving = RIGHT )
THEN
set_Table_r_moving_STOP
     ELSE
SKIP
    END

END
END
```

```
MACHINE   FeedBelt_moving_actuator
SEES datatypes
VARIABLES
      FeedBelt_moving   /* output variable to indicate the state of the belt
wether moving or not */
INVARIANT
      FeedBelt_moving : ONOFF
INITIALISATION
      FeedBelt_moving := ON
OPERATIONS
set_FeedBelt_moving_OFF =
PRE  FeedBelt_moving /= OFF
THEN
FeedBelt_moving := OFF
END;
set_FeedBelt_moving_ON =
PRE  FeedBelt_moving /= ON
THEN
FeedBelt_moving := ON
END
END
```

```
MACHINE   Press_press_state_actuator
SEES datatypes
VARIABLES
      Press_press_state     /* output variable the press state */
INVARIANT
      Press_press_state : ONOFF
INITIALISATION
      Press_press_state := OFF
OPERATIONS
set_Press_press_state_OFF =
PRE  Press_press_state /= OFF
THEN
Press_press_state := OFF
END;
set_Press_press_state_ON =
PRE  Press_press_state /= ON
THEN
Press_press_state := ON
END
END
```

```
MACHINE   Press_v_moving_actuator
SEES datatypes
VARIABLES
      Press_v_moving     /* output variable to determine the dierction of motion
of the press tray */
INVARIANT
      Press_v_moving : VERTICAL_MOTION
INITIALISATION
      Press_v_moving := STOP
OPERATIONS
set_Press_v_moving_UP =
PRE  Press_v_moving /= UP
THEN
Press_v_moving := UP
END;
set_Press_v_moving_STOP =
PRE  Press_v_moving /= STOP
THEN
Press_v_moving := STOP
END;
set_Press_v_moving_DOWN =
PRE  Press_v_moving /= DOWN
THEN
Press_v_moving := DOWN
END
END
```

```
MACHINE   Robot_arm1_magnet_actuator
SEES datatypes
VARIABLES
      Robot_arm1_magnet      /* output variable arm1 magnet */
INVARIANT
      Robot_arm1_magnet : ONOFF
INITIALISATION
      Robot_arm1_magnet := OFF
OPERATIONS
set_Robot_arm1_magnet_OFF =
PRE  Robot_arm1_magnet /= OFF
THEN
Robot_arm1_magnet := OFF
END;
set_Robot_arm1_magnet_ON =
PRE  Robot_arm1_magnet /= ON
THEN
Robot_arm1_magnet := ON
END
END
```

```
MACHINE   Robot_arm1_moving_actuator
SEES datatypes
VARIABLES
      Robot_arm1_moving          /* output variable arm1 motor extend */
INVARIANT
      Robot_arm1_moving : ARM_MOTION
INITIALISATION
      Robot_arm1_moving := STOP
OPERATIONS
set_Robot_arm1_moving_EXTEND =
PRE  Robot_arm1_moving /= EXTEND
THEN
Robot_arm1_moving := EXTEND
```

```
END;
set_Robot_arm1_moving_STOP =
PRE  Robot_arm1_moving /= STOP
THEN
Robot_arm1_moving := STOP
END;
set_Robot_arm1_moving_RETRACT =
PRE  Robot_arm1_moving /= RETRACT
THEN
Robot_arm1_moving := RETRACT
END
END
```

```
MACHINE   Robot_arm2_magnet_actuator
SEES datatypes
VARIABLES
      Robot_arm2_magnet                    /* output variable arm2 magnet */
INVARIANT
      Robot_arm2_magnet : ONOFF
INITIALISATION
      Robot_arm2_magnet := OFF
OPERATIONS
set_Robot_arm2_magnet_OFF =
PRE  Robot_arm2_magnet /= OFF
THEN
Robot_arm2_magnet := OFF
END;
set_Robot_arm2_magnet_ON =
PRE  Robot_arm2_magnet /= ON
THEN
Robot_arm2_magnet := ON
END
END
```

```
MACHINE   Robot_arm2_moving_actuator
SEES datatypes
VARIABLES
      Robot_arm2_moving          /* output variable arm2 motor extending */
INVARIANT
      Robot_arm2_moving : ARM_MOTION
INITIALISATION
      Robot_arm2_moving := STOP
OPERATIONS
set_Robot_arm2_moving_EXTEND =
PRE  Robot_arm2_moving /= EXTEND
THEN
Robot_arm2_moving := EXTEND
END;
set_Robot_arm2_moving_STOP =
PRE  Robot_arm2_moving /= STOP
THEN
Robot_arm2_moving := STOP
END;
set_Robot_arm2_moving_RETRACT =
PRE  Robot_arm2_moving /= RETRACT
THEN
Robot_arm2_moving := RETRACT
END
END
```

```
MACHINE   Robot_r_moving_actuator
SEES datatypes
VARIABLES
      Robot_r_moving  /* output variable: the rotary motion of the robot body */
INVARIANT
      Robot_r_moving : ROTATION
INITIALISATION
      Robot_r_moving := STOP
OPERATIONS
set_Robot_r_moving_RIGHT =
PRE  Robot_r_moving /= RIGHT
THEN
Robot_r_moving := RIGHT
END;
set_Robot_r_moving_STOP =
PRE  Robot_r_moving /= STOP
THEN
Robot_r_moving := STOP
END;
set_Robot_r_moving_LEFT =
PRE  Robot_r_moving /= LEFT
THEN
Robot_r_moving := LEFT
END
END
```

```
MACHINE   Table_r_moving_actuator
SEES datatypes
VARIABLES
      Table_r_moving /* output variable: the direction of the rotation */
INVARIANT
      Table_r_moving : ROTATION
INITIALISATION
      Table_r_moving := STOP
OPERATIONS
set_Table_r_moving_RIGHT =
PRE  Table_r_moving /= RIGHT
THEN
Table_r_moving := RIGHT
END;
set_Table_r_moving_STOP =
PRE  Table_r_moving /= STOP
THEN
Table_r_moving := STOP
END;
set_Table_r_moving_LEFT =
PRE  Table_r_moving /= LEFT
THEN
Table_r_moving := LEFT
END
END
```

```
MACHINE   Table_v_moving_actuator
SEES datatypes
VARIABLES
      Table_v_moving /* output variable: the direction of vertical motion  */
INVARIANT
      Table_v_moving : VERTICAL_MOTION
INITIALISATION
      Table_v_moving := STOP
```

```
OPERATIONS
set_Table_v_moving_UP =
PRE  Table_v_moving /= UP
THEN
Table_v_moving := UP
END;
set_Table_v_moving_STOP =
PRE  Table_v_moving /= STOP
THEN
Table_v_moving := STOP
END;
set_Table_v_moving_DOWN =
PRE  Table_v_moving /= DOWN
THEN
Table_v_moving := DOWN
END
END
```

Bibliography

[Abrial 95] J. R. Abrial, "The B Book: Assigning Programs to Meaning", Cambridge University Press, 1995.

[Aho et al. 86] A. V. Aho, R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[Alford and Burns 76] M. Alford and I. Burns, "R-nets: A Graph Model for Real-Time Software Requirements", Symposium on Computer Software Engineering, New York, Polytechnic Press, 1976, pp97-108.

[Ali 98] M. Ali, "B specification of Steam Boiler", MSC. Thesis Imperial College, September 1998.

[Anderson and Fickas 89] J. S. Anderson and S. Fickas, "A proposed perspective Shift: Viewing Specification Design as a Planning Problem", Proc. IWSSD-5 – 5th Intr. Workshop on software Specification and Design, IEEE, 1989, pp 177-184.

[Antón 95] A. I. Anton, "Goal-Based Requirements Analysis Tool (GBRAT): Requirements Document", Version 0.3, Georgia Institute of Technology Web Page, http://www.cc.gatech.edu/computing/SWEng/Project/reqtsdoc.html , 14 November 1995.

[Antón 96] A. I. Anton, "Goal-Based Requirements Analysis", 2nd IEEE International Conference on Requirements Engineering (ICRE `96), Colorado Springs, Colorado, 15-18 April 1996, pp. 136-144.

[Antón 97] A. I. Anton, ""Goal Identification and Refinement in the specification of Software Based Information Systems", PhD Thesis at Georgia Institute of Technology, June 1997.

[Basili and Weiss 81] V. R. Basili and D. Weiss. "Evaluation of a Software Requirements Document by analysis of change data", in fifth International Conference on Software Engineering, Washiton D.C.: Computer socity Press of IEEE, March 1981, pp 314-23.

[Batory et al. 02] D. Batory, R. E. Lopez-Herrejon, and J. P. Martin, "Generating Product-Lines of Product-Families", Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), 2002.

[B-Core] B Core UK limited (1998), B Toolkit http://www.b-core.com/btoolkit.html.

[Bosch 99] J. Bosch, "Product-line architectures in industry: a case study", International Conference on Software Engineering, Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States, pp. 544 – 554, 1999.

[Bosch 99] J. Bosch, "Product-line architectures in industry: a case study", International Conference on Software Engineering, Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States, pp. 544 – 554, 1999.

[Objectiver] The Objective/Grail Tool: http://www.objectiver.com/download/documents/presentations/KaosCEE-Grail.pdf.

[Bosch 99] J. Bosch, "Product-line architectures in industry: a case study", International Conference on Software Engineering, Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States, pp. 544 – 554, 1999.

[Objectiver] The Objective/Grail Tool: http://www.objectiver.com/download/documents/presentations/KaosCEE-Grail.pdf.

[KAOS] The KAOS method at UCL: http://www.info.ucl.ac.be/research/projects/AVL/ReqEng.html.

[Burstall and Goguen 80] R. Burstall and J. Goguen, "The Semantics of Clear, a Specification Language". Proc. of Advanced Course on Abstract Software Specifications. In Lecture Notes in Computer Science, Springer Verlag, 1980.

[Cholvy at al. 02] L. Cholvy, C. Garion, J. Foisseau, and M. Lemoine, "Requirements Engineering - Are Formal Methods Ready for Industry?", International Workshop on Requirements for High Assurance Systems, 2002.

[Chvalovsky 83] V. Chvalovsky, "Decision tables", Software Practice and Experience 13(1983), pp. 423-29.

[Coulange 98] B. Coulange, "Software Reuse", Translation Editor Iain Craig Springer-Veralg Berlin Heidelberg, 1998.

[Dardenne et al. 93] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition", Science of Computer Programming, Vol. 20, 1993, pp. 3-50.

[Darimont and Van Lamsweerde 96] R. Darimont and A. Van Lamsweerde, "Formal Refinement Patterns for Goal Driven Requirements Elaboration", Proceeding 4th ACM symposium on the foundation of software Engineering (FSE4), San Francisco, Oct 1996, pp. 179-190.

[Darimont et al. 98] R. Darimont, E. Delor, P. Massonet, and A. Van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", IEEE, Proceeding of the 20th International Conference on Software Engineering, Kyoto, April 1998, Vol. 2, pp. 58-62.

[Date 86] C. J. Date, "An Introduction to Database Systems", 4th edition, Addison-Wesley, 1986.

[Davis 93] A. M. Davis, "Software Requirements Objects Functions and States", Prentice Hall PTR, 1993.

[Dingwall-Smith and Finkelstein 03] A. Dingwall-Smith and A. Finkelstein, "Monitoring Goals with Aspects," University College London, Dept. of Computer Science, August 2003.

[Dorf 92] R. C. Dorf, "Modern Control Systems", 6th edition, Addison-Wesley, 1992.

[Dürr and Dusink 93] E. H. Dürr and E. M. Dusink, "Role of VDM++ in the Development of a Real-Time Tracking and Tracing System", In J.C.P. Woodcock and P.G. Larsen (Eds.), *Industrial-Strength Formal Methods*. Springer-Verlag, 1993. Proceedings of FME'93, Odense, Denmark, April 1993.

[Easterbrook 94] S. Easterbrook, "Resolving Requirement Conflicts with Computer-Supported Negotiation", In requirement engineering: social and technical issues, M. Jirotka and J. Goguen (Eds.) Academic Press, 1994, pp. 41-65.

[Easterbrook and Nuseibeh 95] S. M. Easterbrook and B. Nuseibeh, "Managing Inconsistencies in an evolving specification". 2nd IEEE symposium on Requirements Engineering, York, UK, March 1995, pp. 48-55.

[El-Maddah 03] I. A. M. El-Maddah, "Innovating Requirements for Process Control Systems: Ideas Based on Recent Events of RESG ", Requirenautics Quarterly The Newsletter of the Requirements Engineering Specialist Group of the British Computer Society http://www.resg.org.uk, 2003.

[El-Maddah and Maibaum 03a] I. A. M. El-Maddah and T. S. E. Maibaum, 2003, "Goal-Oriented Requirements Analysis for Process Control Systems Design", Proc. 1st ACM and IEEE International Conference on Formal models and methods for co-design MemoCode, France, pp. 45-46.

[El-Maddah and Maibaum 03b] I. A. M. El-Maddah and T. S. E. Maibaum, 2003, "GOPCSD: Goal-oriented Process Control Systems Design", the 12th International FME symposium: Tool Exhibition Notes, Tiziana Margaria (Eds), STAR, Servizio Tecnografico Area della Ricerca del CNR- Pisa Responsibile Salvatore La Polla Luglio 184,41, pp 32-36

[El-Maddah and Maibaum 04a] I. A. M. El-Maddah and T. S. E. Maibaum, "The GOPCSD Tool: An Integrated Development Environment for Process Control Requirements and Design", Fundamental Approaches to Software Engineering FASE04, ETAPS, Spain, 2004.

[El-Maddah and Maibaum 04b] I. A. M. El-Maddah and T. S. E. Maibaum, "Tracing Aspects in Goal driven Requirements of Process Control Systems", Early Aspects 2004 Workshop at The International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, 2004.

[El-Maddah and Maibaum 04c] I. A. M. El-Maddah and T. S. E. Maibaum, "Requirements-Reuse Using GOPCSD: Component-based Development of Process Control Systems", The eighth International conference on software reuse, ICSR8 (in Press), Madrid Spain, 2004.

[Fickas and Helm 92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", IEEE transactions on Software Engineering, June 1992, pp. 470-482.

[Galton 87] A. Galton, "Temporal Logics and their applications", Academic Press Limited, London, 1987.

[Gannon et al. 01] J. D. Gannon, J. M. Purtilo, and M. V. Zelkowitz, "Software Specification: a Comparison of Formal Methods", University of MaryLand, College Park, Maryland, 2001.

[Gilb 03] T. Gilb, "Competitive Engineering, a Handbook for Systems and Software Engineering Management using Planguage", Addison-Wesley, 2003.

[Giorgini et al. 02] P. Giorgini, J. Mylopoulos, E. Nicchiarelli and R. Sebastiani, "Reasoning with Goal Models", Proceedings of the 21st International Conference on Conceptual Modeling (ER2002), Tampere, Finland, October 2002. LNCS - Springer Verlag.

[Goguen and Malcolm Y2K] J. Goguen and G. Malcolm, "Software Engineering with OBJ: Algebraic Specification in Action", 2000; ISBN 0-7923-7757-5. 2000, pages 3-167. The OBJ3 user manual, 1999.

[Goldblatt 92] R. Goldblatt, "Logic of time and computation", 2nd edition, CSLI: Center for the Study of Language and information, Leland Stanford Junior University, 1992.

[GRL] "Goal Driven Requirement Language", Toronto University, Computer Science, http://www.cs.toronto.edu/km/GRL/GRL_introduction.htm.

[Hall 03] A. Hall, "Industrial Experience with Formal Requirements", Praxis Critical Systems, RESG symposium, Imperial College, London, UK, 2003.

[Harel 87] D. Harel, "Statecharts: A Visual Formalization For complex Systems", Science of Computer Programming 8(1987): 231-74.

[Hayes et al. 03] I. J. Hayes, M. A. Jackson, C. B. Jones, "Determining the specification of a control system from that of its environment", proceeding of FME 03, Pisa, Italy, 2003, Keijro Araki, Stefania Gnesi and Dino Mandrioli (Eds.), LNCS 2805, Springer-verlag 2003, pp 154-169.

[Heaven and Finkelstein 03] W. Heaven and A. Finkelstein, "A UML Profile to Support Requirements Engineering with KAOS," IEE Proceedings - Software, 2003.

[Heitmeyer and McLean 83] C. L. Heitmeyer and J. McLean, "Abstract requirements specifications: A new approach and its application". IEEE Trans. Software Eng., SE-9(5):580-589, Sept. 1983.

[Heitmeyer et al. 96] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, "Automated consistency checking of Requirement Specifications", IEEE Transactions on Software Engineering and Methodology, 5(3):231-261, 1996.

[Heitmeyer et al. 98] C. L. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A toolset for specifying and analyzing software requirements". In Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98), Vancouver, Canada, 1998.

[Heninger 80] K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their application", IEEE transactions on Software Engineering 6, January 1980, pp. 2-13.

[Hinchey and Bowen 95] M. G. Hinchey and J. P. Bowen, "Applications of Formal Methods", Prentice Hall, 1995.

[Hunter and Nuseibeh 98] A. Hunter and B. Nuseibeh, "Managing Inconsistent Specifications: reasoning, analysis and action", ACM Transactions on Software Engineering and Methodology , 7(4): 335-367, 1998.

[IEEE 84] Institute of Electrical and Electronic Engineering guide to software Requirements Specification ANSI/IEEE standard 830-1984. New York, 1984.

[Jackson 01] M. Jackson, "Problem Frames: Analyzing and Structuring Software Development problems", Addison Wesley, 2001.

[Jaffe et al. 91] M. S. Jaffe, N. G. Leveson, M. P. E. Heimhdal and B. E. melhart, "Software Requirements Analysis for real-time Process Control Systems", IEEE transactions on Software Engineering 17, 3 (March 1991); pp. 241-58.

[Jones 90] C. B. Jones, "Systematic Software Development using VDM", Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Konrad et al. 02] S. Konrad, L. A. Campbell, and B. H. C. Cheng, "Adding Formal Specifications to Requirements", International Workshop on Requirements for High Assurance Systems, 2002.

[Koymans 92] R. Koymans, "Specifying Message Passing and Time-Critical systems with Temporal Logic", LNCS 6521, Springer-Verlag, 1992.

[Landtsheer et al. 03] R. De Landtsheer, E. Letier and A. Van Lamsweerde, "Deriving Tabular event-based Specifications from goal-oriented Requirements Models", In Proceeding of RE'03, 11th IEEE joint International Requirements Engineering Conference, Monterey (CA) Sept. 2003, pp. 200-210.

[Lano and Haughton 96] K. Lano and H. Haughton, "Specifications in B, An introduction using the B toolkit", Imperial College Press, 1996.

[Lano and Sanchez 97] K. Lano and A. Sanchez, "Design of Real-time Control Systems for event driven Operations", Formal Method Europe, LNCS vol. 1313, Springer-Verlage, Berlin, Germany, 1997.

[Lano et al. Y2Ka] K. Lano, K. Androutsopoulos, and D. Clark, "Structuring and Design of Reactive Systems using RSDS and B", FASE, ETAPS 2000.

[Lano et al. Y2Kb] K. Lano, K. Androutsopoulos and P. Kan, "Structuring Reactive Systems in B AMN", ICFEM 2000.

[Leach 97] Leach, Ronald J., "Software Reuse: Methods, Models and Costs", McGraw-Hill, 1997.

[Leavens and Sitaraman Y2K] G. T. Leavens and M. Sitaraman, "Foundation of Components based Systems", Cambridge, 2000.

[Letier 01] E. Letier, "Reasoning about Agents in Goal-Driven Requirements Engineering", Ph.D. thesis, Catholique de Louvain University, April 2001.

[Letier and van Lamsweerde 02a] E. Letier and A. van Lamsweerde, "High Assurance Requires Goal Orientation", International Workshop on Requirements for High Assurance Systems, 2002.

[Letier and Van Lamsweerde 02b] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", SIGSOFT 2002/FSE-10, Nov. 18-22 Charleston, SC, USA.

[Leveson et al. 94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. Reese, "Requirements Specification for Process Control Systems", IEEE Transactions on Software Engineering, Vol. SE-20, No. 9, pp. 684-707 (September 1994).

[Leveson Y2K] N. G. Leveson, "Completeness in Formal Specification Language Design for Process-Control Systems", FMSP00, Formal Methods in Software Practice, August 2000, Portland.

[Lewerentz and Lindner 95] C. Lewerentz and T. Lindner, "Case Study 'Production Cell' a Comparative Study in Formal Software Development", FZI Karlsruhe, 1995.

[Lewis et al. 98] H. R. Lewis and C. H. Papadimitrus, "Elements of the theory of Computation", Prentice Hall, 1998.

[Liu and Yu 91] L. Liu and E. Yu, "From Requirements to Architectural Design -Using Goals and Scenarios", http://citeseer.nj.nec.com/444188.html.

[Lucas and Van Der Gaag 91] P. Lucas and L. Van Der Gaag, "Principles of Expert Systems Expert Systems", Addison Wesley Publishing Company, 1991.

[Maiden and Sutcliffe 92] N. A. M. Maiden and A. G. Sutcliffe, "Exploiting Reusable Specifications Through Analogy", Communications of the ACM. 34(5), pp 55-64, 1992.

[Mamdani 93] E. H. Mamdani, "Twenty Years of Fuzzy Control: Experiences gained and Lessons Learnt", IEEE International conference on Fuzzy Systems. pp 339-344, 1993.

[Manna and Pnueli 92] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems", Springer-Verlag, 1992.

[Massonet et van Lamsweerde 97] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", Proceeding of RE'97 –3rd IEEE International Symposium on Requirements Engineering, pp. 26-37, 1997.

[Mellor and Ward 85] S. Mellor and P. Ward, "Structured Development for Real-time Systems" (3 volumes), Yourdon Press, 1985.

[Moret 82] B. Moret, "Decision Trees and Diagrams", ACM Computing Surveys, 14, 4 (December 1982): pp. 593-623.

[Mylopoulos Y2K] J. Mylopoulos and J. Castro, "Tropos: A Framework for Requirements-Driven Software Development", J. Brinkkemper and A. Solvberg (eds.), Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer-Verlag, p. 261-273, June 2000.

[Nielmela and Ihme 01] E. Nielmela, T. Ihme ,"Product Line Software Engineering of Embedded Systems", Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, Toronto, Ontario, Canada, pp. 118 – 125, 2001.

[Nord Y2K] R. L. Nord, "Meeting the Product-Line Goals for an Embedded Real-time System", Software Architectures for Product Families. International Workshop IW-SAPF-3. Proceedings (Lecture Notes in Computer Science Vol.1951). Springer-Verlag. 2000. Berlin, Germany. pp. 19-29.

[Nuseibeh and Easterbrook Y2K] B. Nuseibeh and S. Easterbrook, "Requirements Engineering: a Roadmap", Future of Software Engineering, Limerick, Ireland, 2000.

[Parnas and Madey 95] David Lorge Parnas and Jan Madey, "Functional Documents of Critical Systems", Science of Computer Programming, 25(1): 41-61, October 1995.

[Peterson 77] J. Peterson, "Petri Nets", ACM computing Surveys 9, 3 (September 1977): pp. 223-52.

[Petri 62] A. C. Petri, "Kommikation mit Automaten", PhD. Diss. University of Bonn, Bonn, Germany 1962.

[Piveropoulos Y2K] M. Piveropoulos, "Requirements Engineering for Hard Real-Time Systems", PhD Thesis, The University of York, 2000.

[Potts 95] C. Potts, "Using Schematic Scenario to Understand User Needs", Proc. DIS'95, ACM Symposium on designing interactive systems: Processes, Practices and Techniques, University of Michigan, August 1995.

[Pressman 92] R. S. Pressman, "Software Engineering: A Practitioner's Approach" 3rd edition 1992, McGRAW-Hill international edition.

[Reubenstein and Waters 91] H. B. Reubenstein and R. C. Waters, "The Requirements Apprentice: Automated Assistance of Requirement Acquisition", IEEE Transactions on Software Engineering, Vol17 No. 3, March 1991, pp. 226-240.

[Robertson and Robertson 99] S. Robertson and J. Robertson, "Mastering the Requirements Process", ACM Press, 1999.

[Rockstrom 82] A. Rockstrom and R. Saracoo, "SDL-CCITT Specification and Description Language", IEEE transactions on Communication 30, 6 (June 1982): pp. 1310-18.

[Rushby 95] J. Rushby, "Formal Methods and their Role in the Certification of Critical Systems", URL = http://www.csl.sri.com/papers/csl-95-1/.

[Sekerinski 98] E. Sekerinski, "Graphical Design of Reactive Systems", D. Bert (Ed.) 2nd International B conference, Montpellier, France, Springer-Verlag, 1998.

[Sommerville 01] I. Sommerville, "Software Engineering", 6th edition, Addison Wesley Publisher, 2001.

[Spivey 92] J. M. Spivey, "The Z Notation: A Reference Manual", 2nd edition, Hermel Hempstead: Prentice-Hall, 1992.

[Storey 96] N. Storey, "Safety Critical Computer Systems", Addison-Wesley Longman, 1996.

[Takiguhi 01] K. Takiguhi, "Recent advances in PLC functional devices," LEOS SUM '01 (Copper Mt., CO), Paper MD2.1, pp. 9-10, 2001.

[Turner and McCluskey 94] J. G. Turner and T. L. McCluskey, "The construction of Formal Specifications An introduction to Model-based and Algebraic Approaches", McGraw-Hill Book Company, 1994.

[VanLamsweerde et al. 91] A. Van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, "The KAOS Project: Knowledge acquisition in automated specifications of software", proceeding AAAI Spring Symposium series, Track: "Design of composite systems", Stanford University, March 1991, pp 59-62.

[VanLamsweerde et al. 95] A. Van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed elaboration of Requirements for meeting scheduler: Problems and Lessons learned", Proc. RE'95- 2nd intr. Symp. on Requirement Engineering, York, IEEE, 1995.

[VanLamsweerde et al. 98a] A. Van Lamsweerde and E. Letier, "Obstacles in Goal-driven Requirement Engineering", Proceeding ICSE'98 20th international conference on software engineering, Kyoto, ACM-IEEE, April 1998.

[VanLamsweerde et al. 98b] A. Van Lamsweerde, R. Darimont and E. Letier "Managing Conflicts in Goal-Driven Requirement Engineering", IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, Nov. 1998.

[VanLamsweerde Y2K] A. Van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective" invited paper for ICSE'2000 Proc. 22nd International Conference on Software Engineering, Limerick, ACM Press, June 2000.

[Wieringa 01] R. J. Wieringa, "Design Methods for Reactive Systems: Yourdan, Statemate and the UML", Morgan Kufmann Publisher, San Francisco, 2001.

[Winston 92] P. H. Winston, "Artificial Intelligence", Addison Wesley Publishing company, 1992.

[Winter 01] K. Winter, "Model Checking Abstract State Machines", PhD thesis, Technischen University, Berlin, 2001.

[Wordsworth 92] J. B. Wordsworth, "Software Development with Z", Addison-Wesley, 1992.

[Wordsworth 96] J. B. Wordsworth, "Software Engineering with B", Addison-Wesley, 1996.

[Yu and Mylopoulos 98] E. Yu and J. Mylopoulos, "Why Goal-Oriented Requirements Engineering", Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality. June 1998, Pisa, Italy. E. Dubois, A. L. Opdahl, K. Pohl, eds. Presses Universitaires de Namur, 1998. pp. 15-22.

[Zadeh 84] L. A. Zadeh, "Making Computer thinks like People", IEEE spectrum 8/1994, pp. 26-32.

[Zave 97] P. Zave, "Classification of Research Effort in Requirements Engineering", ACM computing surveys, 29(4): pp. 315-321, 1997.

[Zave and Jackson 97] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", ACM Trans. Software Eng. And Methodology, 6(1). 1997. 85.

[Zorzo et al. 99] A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, I.S. Welch, "Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study", Software Practice & Experience. , Vol. 29, No. 7, 1999, pp. 1-21.